About the Author

Herbert Schildt is a leading authority on the Java, C++, C, and C# languages. His programming books have sold millions of copies worldwide and have been translated into all major foreign languages. He is the author of numerous books on Java, including Java: A Beginner’s Guide, Herb Schildt’s Java Programming Cookbook, Swing: A Beginner’s Guide, and The Art of Java. Among his other bestsellers are C++: The Complete Reference™, C#: The Complete Reference™, and C: The Complete Reference™. Although interested in all facets of computing, his primary focus is computer languages, including compilers, interpreters, and robotic control languages. He also has an active interest in the standardization of languages. Schildt holds both graduate and undergraduate degrees from the University of Illinois. He can be reached at his consulting office at (217) 586-4683. His web site is www.HerbSchildt.com.

About the Technical Editor
Dr. Danny Coward has been a contributor to the Java Platforms since 1997. He was a founding member of the Java EE group while at Sun. He has served as a member of the Java Community Process Executive Committee and has been a leading contributor to all editions of the Java Platform—Java SE, Java ME, and Java EE—and established the original JavaFX team.
LICENSORS MAKE NO GUARanteES OR WARRantees AS TO THE ACCURAcY, ADEQUAcy OR COMPLETEnESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRAntY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRAntIES OF MERCHAntABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.
Contents at a Glance

Part I The Java Language

1. The History and Evolution of Java
2. An Overview of Java
3. Data Types, Variables, and Arrays
4. Operators
5. Control Statements
6. Introducing Classes
7. A Closer Look at Methods and Classes
8. Inheritance
9. Packages and Interfaces
10. Exception Handling
11. Multithreaded Programming
Contents

Preface

Part I  The Java Language

Chapter 1  The History and Evolution of Java

Java’s Lineage

The Birth of Modern Programming: C

C++: The Next Step

The Stage Is Set for Java

The Creation of Java

The C# Connection

How Java Changed the Internet

Java Applets

Security

Portability
Java’s Magic: The Bytecode

Servlets: Java on the Server Side

The Java Buzzwords

Simple

Object-Oriented

Robust

Multithreaded

Architecture-Neutral

Interpreted and High Performance

Distributed

Dynamic

The Evolution of Java

Java SE 7

A Culture of Innovation

Chapter 2    An Overview of Java

Object-Oriented Programming
Two Paradigms

Abstraction

The Three OOP Principles

A First Simple Program

Entering the Program

Compiling the Program

A Closer Look at the First Sample Program

A Second Short Program

Two Control Statements

The if Statement

The for Loop

Using Blocks of Code

Lexical Issues

Whitespace

Identifiers
Java Is a Strongly Typed Language

The Primitive Types

Integers

byte

short

int

long

Floating-Point Types

float

double
Characters

Booleans

A Closer Look at Literals

Integer Literals

Floating-Point Literals

Boolean Literals

Character Literals

String Literals

Variables

Declaring a Variable

Dynamic Initialization

The Scope and Lifetime of Variables

Type Conversion and Casting

Java’s Automatic Conversions

Casting Incompatible Types
Automatic Type Promotion in Expressions

The Type Promotion Rules

Arrays

One-Dimensional Arrays

Multidimensional Arrays

Alternative Array Declaration Syntax

A Few Words About Strings

A Note to C/C++ Programmers About Pointers

Chapter 4 Operators

Arithmetic Operators

The Basic Arithmetic Operators

The Modulus Operator

Arithmetic Compound Assignment
Operators

Increment and Decrement

The Bitwise Operators

The Bitwise Logical Operators

The Left Shift

The Right Shift

The Unsigned Right Shift

Bitwise Operator Compound Assignments

Relational Operators

Boolean Logical Operators

Short-Circuit Logical Operators

The Assignment Operator

The ? Operator

Operator Precedence

Using Parentheses
Chapter 5  Control Statements

Java’s Selection Statements

if

switch

Iteration Statements

while

do-while

for

The For-Each Version of the for Loop

Nested Loops

Jump Statements

Using break

Using continue

Chapter 6  Introducing Classes

Class Fundamentals
The General Form of a Class

A Simple Class

Declaring Objects

A Closer Look at new

Assigning Object Reference Variables

Introducing Methods

Adding a Method to the Box Class

Returning a Value

Adding a Method That Takes Parameters

Constructors

Parameterized Constructors

The this Keyword

Instance Variable Hiding

Garbage Collection

The finalize() Method
Chapter 7  A Closer Look at Methods and Classes

Overloading Methods

Overloading Constructors

Using Objects as Parameters

A Closer Look at Argument Passing

Returning Objects

Recursion

Introducing Access Control

Understanding static

Introducing final

Arrays Revisited

Introducing Nested and Inner Classes

Exploring the String Class

Using Command-Line Arguments

Varargs: Variable-Length Arguments
Chapter 8     Inheritance

Inheritance Basics

Member Access and Inheritance

A More Practical Example

A Superclass Variable Can Reference a Subclass Object

Using super

Using super to Call Superclass Constructors

A Second Use for super

Creating a Multilevel Hierarchy

When Constructors Are Called

Method Overriding

Dynamic Method Dispatch
Why Overridden Methods?

Applying Method Overriding

Using Abstract Classes

Using final with Inheritance

Using final to Prevent Overriding

Using final to Prevent Inheritance

The Object Class

Chapter 9 Packages and Interfaces

Packages

Defining a Package

Finding Packages and CLASSPATH

A Short Package Example

Access Protection

An Access Example

Importing Packages

Interfaces
Chapter 10 Exception Handling

Exception-Handling Fundamentals

Exception Types

Uncaught Exceptions

Using try and catch

Displaying a Description of an Exception

Multiple catch Clauses

Nested try Statements

throw
Java’s Built-in Exceptions

Creating Your Own Exception Subclasses

Chained Exceptions

Three New JDK 7 Exception Features

Using Exceptions

Chapter 11  Multithreaded Programming

The Java Thread Model

Thread Priorities

Synchronization

Messaging

The Thread Class and the Runnable Interface

The Main Thread

Creating a Thread
Implementing Runnable
Extending Thread
Choosing an Approach
Creating Multiple Threads
Using isAlive() and join()
Thread Priorities
Synchronization
Using Synchronized Methods
The synchronized Statement
Interthread Communication
Deadlock
Suspending, Resuming, and Stopping Threads
Suspending, Resuming, and Stopping Threads Using Java 1.1 and Earlier
The Modern Way of Suspending, Resuming, and Stopping Threads

Obtaining A Thread’s State

Using Multithreading

Chapter 12  Enumerations, Autoboxing, and Annotations (Metadata)

Enumerations

Enumeration Fundamentals

The values( ) and valueOf( ) Methods

Java Enumerations Are Class Types

Enumerations Inherit Enum

Another Enumeration Example

Type Wrappers

Character

Boolean
The Numeric Type Wrappers

Autoboxing

Autoboxing and Methods

Autoboxing/Unboxing Occurs in Expressions

Autoboxing/Unboxing Boolean and Character Values

Autoboxing/Unboxing Helps Prevent Errors

A Word of Warning

Annotations (Metadata)

Annotation Basics

Specifying a Retention Policy

Obtaining Annotations at Run Time by Use of Reflection

The AnnotatedElement Interface
Chapter 13  I/O, Applets, and Other Topics

I/O Basics

Streams

Byte Streams and Character Streams

The Predefined Streams

Reading Console Input

Reading Characters

Reading Strings

Writing Console Output

The PrintWriter Class
Reading and Writing Files

Automatically Closing a File

Applet Fundamentals

The transient and volatile Modifiers

Using instanceof

strictfp

Native Methods

Problems with Native Methods

Using assert

Assertion Enabling and Disabling

Options

Static Import

Invoking Overloaded Constructors Through this()

Chapter 14 Generics

What Are Generics?

A Simple Generics Example
Generics Work Only with Objects

Generic Types Differ Based on Their Type Arguments

How Generics Improve Type Safety

A Generic Class with Two Type Parameters

The General Form of a Generic Class

Bounded Types

Using Wildcard Arguments

Bounded Wildcards

Creating a Generic Method

Generic Constructors

Generic Interfaces

Raw Types and Legacy Code

Generic Class Hierarchies

Using a Generic Superclass
A Generic Subclass

Run-Time Type Comparisons

Within a Generic Hierarchy

Casting

Overriding Methods in a Generic Class

Type Inference with Generics

Erasure

Bridge Methods

Ambiguity Errors

Some Generic Restrictions

Type Parameters Can’t Be Instantiated

Restrictions on Static Members

Generic Array Restrictions

Generic Exception Restriction
Chapter 15  String Handling

The String Constructors

String Length

Special String Operations

String Literals

String Concatenation

String Concatenation with Other Data Types

String Conversion and toString()

Character Extraction

charAt()

getChars()

getBytes()

toCharArray()

String Comparison
equals( ) and equalsIgnoreCase( )

regionMatches( )

startsWith( ) and endsWith( )

equals( ) Versus ==

c compareTo( )

Searching Strings

Modifying a String

substring( )

concat( )

replace( )

trim( )

Data Conversion Using valueOf( )

Changing the Case of Characters Within a String

Additional String Methods

StringBuffer
StringBuffer Constructors

length() and capacity()

ensureCapacity()

setLength()

charAt() and setCharAt()

getChars()

append()

insert()

reverse()

delete() and deleteCharAt()

replace()

substring()

Additional StringBuffer Methods

StringBuilder

Chapter 16  Exploring java.lang
Primitive Type Wrappers

Number

Double and Float

Understanding `isInfinite()` and `isNaN()`

Byte, Short, Integer, and Long

Character

Additions to Character for Unicode Code Point Support

Boolean

Void

Process

Runtime

Memory Management

Executing Other Programs

ProcessBuilder
Using `currentTimeMillis()` to Time Program Execution

Using `arraycopy()`

Environment Properties

Object

Using `clone()` and the Cloneable Interface

Class

ClassLoader

Math

Trigonometric Functions

Exponential Functions

Rounding Functions

Miscellaneous Math Methods

StrictMath
Compiler

Thread, ThreadGroup, and Runnable

The Runnable Interface

Thread

ThreadGroup

ThreadLocal and InheritableThreadLocal

Package

RuntimePermission

Throwable

SecurityManager

StackTraceElement

Enum

ClassValue

The CharSequence Interface

The Comparable Interface

The Appendable Interface
Chapter 17 java.util Part 1: The Collections Framework

Collections Overview

JDK 5 Changed the Collections Framework
Generics Fundamentally Changed the Collections Framework

Autoboxing Facilitates the Use of Primitive Types

The For-Each Style for Loop

The Collection Interfaces

The Collection Interface

The List Interface

The Set Interface

The SortedSet Interface

The NavigableSet Interface

The Queue Interface

The Deque Interface

The Collection Classes

The ArrayList Class

The LinkedList Class
The HashSet Class
The LinkedHashSet Class
The TreeSet Class
The PriorityQueue Class
The ArrayDeque Class
The EnumSet Class

Accessing a Collection via an Iterator

Using an Iterator

The For-Each Alternative to Iterators

Storing User-Defined Classes in Collections

The RandomAccess Interface

Working with Maps

The Map Interfaces

The Map Classes
Comparators

Using a Comparator

The Collection Algorithms

Arrays

Why Generic Collections?

The Legacy Classes and Interfaces

The Enumeration Interface

Vector

Stack

Dictionary

Hashtable

Properties

Using store() and load()

Parting Thoughts on Collections

Chapter 18  java.util Part 2: More Utility Classes

StringTokenizer
BitSet

Date

Calendar

GregorianCalendar

TimeZone

SimpleTimeZone

Locale

Random

Observable

The Observer Interface

An Observer Example

Timer and TimerTask

Currency

Formatter

The Formatter Constructors

The Formatter Methods
Formatting Basics

Formatting Strings and Characters

Formatting Numbers

Formatting Time and Date

The %n and %%% Specifiers

Specifying a Minimum Field Width

Specifying Precision

Using the Format Flags

Justifying Output

The Space, +, 0, and ( Flags

The Comma Flag

The # Flag

The Uppercase Option

Using an Argument Index
Closing a Formatter

The Java printf() Connection

Scanner

The Scanner Constructors

Scanning Basics

Some Scanner Examples

Setting Delimiters

Other Scanner Features

The ResourceBundle, ListResourceBundle, ___________ and PropertyResourceBundle Classes

Miscellaneous Utility Classes and Interfaces

The java.util Subpackages

java.util.concurrent, java.util.concurrent.atomic, ___________ and java.util.concurrent.locks
Chapter 19  Input/Output: Exploring java.io

The I/O Classes and Interfaces

File

Directories

Using FilenameFilter

The listFiles( ) Alternative

Creating Directories

The AutoCloseable, Closeable, and Flushable Interfaces

I/O Exceptions
Two Ways to Close a Stream

The Stream Classes

The Byte Streams

InputStream

OutputStream

FileInputStream

FileOutputStream

ByteArrayInputStream

ByteArrayOutputStream

Filtered Byte Streams

Buffered Byte Streams

SequenceInputStream

PrintStream

DataOutputStream and DataInputStream

RandomAccessFile
The Character Streams

Reader

Writer

FileReader

FileWriter

CharArrayReader

CharArrayWriter

BufferedReader

BufferedWriter

PushbackReader

PrintWriter

The Console Class

Serialization

Serializable

Externalizable
Chapter 20 Exploring NIO

The NIO Classes

NIO Fundamentals

Buffers

Channels

Charsets and Selectors

Enhancements Added to NIO by JDK 7

The Path Interface

The Files Class

The Paths Class
The File Attribute Interfaces

The FileSystem, FileSystems, and FileStore Classes

Using the NIO System

Use NIO for Channel-Based I/O

Use NIO for Stream-Based I/O

Use NIO for Path and File System Operations

Pre-JDK 7 Channel-Based Examples

Read a File, Pre-JDK 7

Write to a File, Pre-JDK 7

Chapter 21 Networking

Networking Basics

The Networking Classes and Interfaces

InetAddress

Factory Methods
Chapter 22  The Applet Class

Two Types of Applets

Applet Basics
The Applet Class

Applet Architecture

An Applet Skeleton

Applet Initialization and Termination

Overriding update( )

Simple Applet Display Methods

Requesting Repainting

A Simple Banner Applet

Using the Status Window

The HTML APPLET Tag

Passing Parameters to Applets

Improving the Banner Applet

getDocumentBase( ) and getCodeBase( )

AppletContext and showDocument( )

The AudioClip Interface
Chapter 23  Event Handling

Two Event Handling Mechanisms

The Delegation Event Model

Events

Event Sources

Event Listeners

Event Classes

The ActionEvent Class

The AdjustmentEvent Class

The ComponentEvent Class

The ContainerEvent Class

The FocusEvent Class

The InputEvent Class

The ItemEvent Class
The KeyEvent Class

The MouseEvent Class

The MouseWheelEvent Class

The TextEvent Class

The WindowEvent Class

Sources of Events

Event Listener Interfaces

The ActionListener Interface

The AdjustmentListener Interface

The ComponentListener Interface

The ContainerListener Interface

The FocusListener Interface

The ItemListener Interface

The KeyListener Interface

The MouseListener Interface
The **MouseMotionListener** Interface

The **MouseWheelListener** Interface

The **TextListener** Interface

The **WindowFocusListener** Interface

The **WindowListener** Interface

Using the Delegation Event Model

Handling Mouse Events

Handling Keyboard Events

**Adapter Classes**

**Inner Classes**

**Anonymous Inner Classes**

Chapter 24 Introducing the AWT: Working with Windows, Graphics, and Text

**AWT Classes**
Window Fundamentals

Component

Container

Panel

Window

Frame

Canvas

Working with Frame Windows

Setting the Window’s Dimensions

Hiding and Showing a Window

Setting a Window’s Title

Closing a Frame Window

Creating a Frame Window in an Applet

Handling Events in a Frame Window

Creating a Windowed Program
Displaying Information Within a Window

Working with Graphics

- Drawing Lines
- Drawing Rectangles
- Drawing Ellipses and Circles
- Drawing Arcs
- Drawing Polygons

Sizing Graphics

Working with Color

- Color Methods
- Setting the Current Graphics Color

A Color Demonstration Applet

Setting the Paint Mode

Working with Fonts

Determining the Available Fonts
Creating and Selecting a Font

Obtaining Font Information

Managing Text Output Using FontMetrics

Displaying Multiple Lines of Text

Centering Text

Multiline Text Alignment

Chapter 25 Using AWT Controls, Layout Managers, and Menus

Control Fundamentals

Adding and Removing Controls

Responding to Controls

The HeadlessException

Labels

Using Buttons

Handling Buttons
Applying Check Boxes

Handling Check Boxes

CheckboxGroup

Choice Controls

Handling Choice Lists

Using Lists

Handling Lists

Managing Scroll Bars

Handling Scroll Bars

Using a TextField

Handling a TextField

Using a TextArea

Understanding Layout Managers

FlowLayout

BorderLayout

Using Insets
GridLayout
CardLayout
GridBagLayout

Menu Bars and Menus

Dialog Boxes
FileDialog

Handling Events by Extending AWT Components

Extending Button

Extending Checkbox

Extending a Check Box Group

Extending Choice

Extending List

Extending Scrollbar

A Word About Overriding paint( )

Chapter 26 Images
File Formats

Image Fundamentals: Creating, Loading, and Displaying

Creating an Image Object

Loading an Image

Displaying an Image

ImageObserver

Double Buffering

MediaTracker

ImageProducer

MemoryImageSource

ImageConsumer

PixelGrabber

ImageFilter

CropImageFilter

RGBImageFilter
Chapter 27  The Concurrency Utilities

The Concurrent API Packages

java.util.concurrent

java.util.concurrent.atomic

java.util.concurrent.locks

Using Synchronization Objects

Semaphore

CountDownLatch

CyclicBarrier

Exchanger

Phaser

Using an Executor

A Simple Executor Example

Using Callable and Future
The TimeUnit Enumeration

The Concurrent Collections

Locks

Atomic Operations

Parallel Programming via the Fork/Join Framework

The Main Fork/Join Classes

The Divide-and-Conquer Strategy

A Simple First Fork/Join Example

Understanding the Impact of the Level of Parallelism

An Example that Uses RecursiveTask<V>

Executing a Task Asynchronously

Cancelling a Task

Determining a Task’s Completion Status
Restarting a Task

Things to Explore

Some Fork/Join Tips

The Concurrency Utilities Versus Java’s Traditional Approach

Chapter 28 Regular Expressions and Other Packages

The Core Java API Packages

Regular Expression Processing

Pattern

Matcher

Regular Expression Syntax

Demonstrating Pattern Matching

Two Pattern-Matching Options

Exploring Regular Expressions

Reflection

Remote Method Invocation (RMI)
Part III  Software Development Using Java

Chapter 29  Java Beans

What Is a Java Bean?

Advantages of Java Beans

Introspection

Design Patterns for Properties

Design Patterns for Events

Methods and Design Patterns

Using the BeanInfo Interface

Bound and Constrained Properties

Persistence
Chapter 30  Introducing Swing

The Origins of Swing

Swing Is Built on the AWT

Two Key Swing Features

Swing Components Are Lightweight

Swing Supports a Pluggable Look and Feel

The MVC Connection
Components and Containers

Components

Containers

The Top-Level Container Panes

The Swing Packages

A Simple Swing Application

Event Handling

Create a Swing Applet

Painting in Swing

Painting Fundamentals

Compute the Paintable Area

A Paint Example

Chapter 31 Exploring Swing

JLabel and ImageIcon

JTextField

The Swing Buttons
Chapter 32  Servlets

Background

The Life Cycle of a Servlet

Servlet Development Options

Using Tomcat

A Simple Servlet
Create and Compile the Servlet Source Code

Start Tomcat

Start a Web Browser and Request the Servlet

The Servlet API

The javax.servlet Package

The Servlet Interface

The ServletConfig Interface

The ServletContext Interface

The ServletRequest Interface

The ServletResponse Interface

The GenericServlet Class

The ServletInputStream Class

The ServletOutputStream Class

The Servlet Exception Classes
Reading Servlet Parameters

The javax.servlet.http Package

The HttpServletRequest Interface

The HttpServletResponse Interface

The HttpSession Interface

The HttpSessionBindingListener Interface

The Cookie Class

The HttpServlet Class

The HttpSessionEvent Class

The HttpSessionBindingEvent Class

Handling HTTP Requests and Responses

Handling HTTP GET Requests

Handling HTTP POST Requests

Using Cookies
Chapter 33  Financial Applets and Servlets

Finding the Payments for a Loan

The RegPay Fields

The init( ) Method

The makeGUI( ) Method

The actionPerformed( ) Method

The compute( ) Method

Finding the Future Value of an Investment

Finding the Initial Investment Required to Achieve a Future Value

Finding the Initial Investment Needed for a Desired Annuity

Finding the Maximum Annuity for a Given Investment
Finding the Remaining Balance on a Loan

Creating Financial Servlets

Converting the RegPay Applet into a Servlet

The RegPayS Servlet

Some Things to Try

Chapter 34 Creating a Download Manager in Java

Understanding Internet Downloads

An Overview of the Download Manager

The Download Class

The Download Variables

The Download Constructor

The download( ) Method

The run( ) Method

The stateChanged( ) Method
Action and Accessor Methods

The ProgressRenderer Class

The DownloadsTableModel Class

The addDownload( ) Method

The clearDownload( ) Method

The getColumnClass( ) Method

The getValueAt( ) Method

The update( ) Method

The DownloadManager Class

The DownloadManager Variables

The DownloadManager Constructor

The verifyUrl( ) Method

The tableSelectionChanged( ) Method

The updateButtons( ) Method
Handling Action Events

Compiling and Running the Download Manager

Enhancing the Download Manager

Appendix Using Java’s Documentation Comments

The javadoc Tags

@author

{@code}

@deprecated

{@docRoot}

@exception

{@inheritDoc}

{@link}

{@linkplain}

{@literal}

@param
The General Form of a Documentation Comment

What javadoc Outputs

An Example that Uses Documentation Comments

Index
Preface

Java is one of the world’s most important and widely used computer languages. Furthermore, it has held that distinction for many years. Unlike some other computer languages whose influence has waned with the passage of time, Java’s has grown stronger. Java leapt to the forefront of Internet programming with its first release. Each subsequent version has solidified that position. Today, it is still the first and best choice for developing web-based applications. Java is also part of the smartphone revolution because it is used for Android programming. Simply put: much of the modern world runs on Java code. Java really is that important.

A key reason for Java’s success is its agility. Since its original 1.0 release, Java has continually adapted to changes in the programming environment and to changes in the way that programmers program. Most importantly, it has not just followed the trends, it has helped create them. Java’s ability to accommodate the
fast rate of change in the computing world is a crucial part of why it has been and continues to be so successful.

Since this book was first published in 1996, it has gone through several editions, each reflecting the ongoing evolution of Java. This is the Eighth edition, and it has been updated for Java SE 7. As a result, it contains a substantial amount of new material. For example, it includes coverage of the Project Coin language enhancements, the expanded features of NIO (NIO.2), and the Fork/Join Framework. In general, discussions of the new features are integrated into existing chapters, but because of the many additions to NIO, it is now discussed in its own chapter. However, the overall structure of the book remains the same. This means that if you are familiar with the previous edition, you will feel right at home with this version.

A Book for All Programmers

This book is for all programmers, whether you are a novice or an experienced pro. The beginner will find its carefully paced discussions and many examples
especially helpful. Its in-depth coverage of Java’s more advanced features and libraries will appeal to the pro. For both, it offers a lasting resource and handy reference.

What’s Inside
This book is a comprehensive guide to the Java language, describing its syntax, keywords, and fundamental programming principles. Significant portions of the Java API library are also examined. The book is divided into four parts, each focusing on a different aspect of the Java programming environment.

Part I presents an in-depth tutorial of the Java language. It begins with the basics, including such things as data types, operators, control statements, and classes. It then moves on to inheritance, packages, interfaces, exception handling, and multithreading. The final chapters in Part I describe annotations, enumerations, autoboxing, and generics. I/O and applets are also introduced.

Part II examines key aspects of Java’s standard API library. Topics include strings, I/O, networking, the
standard utilities, the Collections Framework, applets, GUI-based controls, imaging, and concurrency (including the new Fork/Join Framework).

**Part III** looks at three important Java technologies: Java Beans, servlets, and Swing.

**Part IV** contains two chapters that show examples of Java in action. The first chapter develops several applets that perform various popular financial calculations, such as computing the regular payment on a loan or the minimum investment needed to withdraw a desired monthly annuity. This chapter also shows how to convert those applets into servlets. The second chapter develops a download manager that oversees the downloading of files. It includes the ability to start, stop, and resume a transfer. Both chapters are adapted from my book The Art of Java, which I co-authored with James Holmes.

Don’t Forget: Code on the Web
Remember, the source code for all of the examples in this book is available free-of-charge on the Web at www.oraclepressbooks.com.
Special Thanks

I want to give special thanks to Patrick Naughton, Joe O’Neil, James Holmes, and Danny Coward.

Patrick Naughton was one of the creators of the Java language. He also helped write the first edition of this book. For example, among many other contributions, much of the material in Chapters 19, 21, and 26 was initially provided by Patrick. His insights, expertise, and energy contributed greatly to the success of that book.

During the preparation of the second and third editions of this book, Joe O’Neil provided initial drafts for the material now found in Chapters 28, 29, 31, and 32 of this edition. Joe helped on several of my books and his input has always been top-notch.

James Holmes provided Chapter 34. James is an extraordinary programmer and author. He was my co-author on The Art of Java and is the author of Struts: The Complete Reference™, and a co-author of JSF: The Complete Reference™.

Danny Coward is the technical editor for this edition.
of the book. His advice, insights, and suggestions were of great value and much appreciated.

HERBERT SCHILDT

For Further Study

Java: The Complete Reference™ is your gateway to the Herb Schildt series of programming books. Here are some others that you will find of interest.

To learn more about Java programming, we recommend the following:

Herb Schildt’s Java Programming Cookbook

Java: A Beginner’s Guide

Swing: A Beginner’s Guide

The Art Of Java

To learn about C++, you will find these books especially helpful:

C++: The Complete Reference™

Herb Schildt’s C++ Programming Cookbook
To learn about C#, we suggest the following Schildt books:

- C#: The Complete Reference™
- C#: A Beginner’s Guide

To learn about the C language, the following title will be of interest:

- C: The Complete Reference™

When you need solid answers, fast, turn to Herbert Schildt, the recognized authority on programming.
PART I
The Java Language

CHAPTER 1
The History and Evolution of Java

CHAPTER 2
An Overview of Java

CHAPTER 3
Data Types, Variables, and Arrays

CHAPTER 4
Operators

CHAPTER 5
Control Statements

CHAPTER 6
Introducing Classes

CHAPTER 7
A Closer Look at Methods and Classes

CHAPTER 8
CHAPTER 1
The History and Evolution of Java

To fully understand Java, one must understand the reasons behind its creation, the forces that shaped it, and the legacy that it inherits. Like the successful computer languages that came before, Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission. While the remaining chapters of this book describe the practical aspects of Java—including its syntax, key libraries, and applications—this chapter explains how and why Java came about, what makes it so important, and how it has evolved over the years.

Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language. Computer language innovation and development occurs for two fundamental reasons:

• To adapt to changing environments and uses
• To implement refinements and improvements in the art of programming

As you will see, the development of Java was driven by both elements in nearly equal measure.

Java’s Lineage
Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java’s object-oriented features were influenced by C++. In fact, several of Java’s defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades. For these reasons, this section reviews the sequence of events and forces that led to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.
The Birth of Modern Programming: C

The C language shook the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs. As you probably know, when a computer language is designed, trade-offs are often made, such as the following:

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn’t very powerful, and its lack of
structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult.

Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles. Instead, they relied upon the GOTO as a primary means of program control. As a result, programs written using these languages tended to produce “spaghetti code”—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand. While languages like Pascal are structured, they were not designed for efficiency, and failed to include certain features necessary to make them applicable to a wide range of programs. (Specifically, given the standard dialects of Pascal available at the time, it was not practical to consider using Pascal for systems-level code.)
So, just prior to the invention of C, no one language had reconciled the conflicting attributes that had dogged earlier efforts. Yet the need for such a language was pressing. By the early 1970s, the computer revolution was beginning to take hold, and the demand for software was rapidly outpacing programmers’ ability to produce it. A great deal of effort was being expended in academic circles in an attempt to create a better computer language. But, and perhaps most importantly, a secondary force was beginning to be felt. Computer hardware was finally becoming common enough that a critical mass was being reached. No longer were computers kept behind locked doors. For the first time, programmers were gaining virtually unlimited access to their machines. This allowed the freedom to experiment. It also allowed programmers to begin to create their own tools. On the eve of C’s creation, the stage was set for a quantum leap forward in computer languages.

Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an
older language called BCPL, developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s. For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

The creation of C is considered by many to have marked the beginning of the modern age of computer languages. It successfully synthesized the conflicting attributes that had so troubled earlier languages. The result was a powerful, efficient, structured language that was relatively easy to learn. It also included one other, nearly intangible aspect: it was a programmer’s language. Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C is different. It was designed, implemented, and developed by real,
working programmers, reflecting the way that they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language. The result was a language that programmers liked to use. Indeed, C quickly attracted many followers who had a near-religious zeal for it. As such, it found wide and rapid acceptance in the programmer community. In short, C is a language designed by and for programmers. As you will see, Java inherited this legacy.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is
fundamental to the creation of C++, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The first widespread language was, of course, FORTRAN. While FORTRAN was an impressive first step, it is hardly a language that encourages clear and easy-to-understand programs. The 1960s gave birth to structured programming. This is the method of
programming championed by languages such as C. The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called object-oriented programming (OOP). Object-oriented programming is discussed in detail later in this book, but here is a brief definition: OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.

In the final analysis, although C is one of the world’s great programming languages, there is a limit to its ability to handle complexity. Once the size of a program exceeds a certain point, it becomes so complex that it is difficult to grasp as a totality. While the precise size at which this occurs differs, depending upon both the
nature of the program and the programmer, there is always a threshold at which a program becomes unmanageable. C++ added features that enabled this threshold to be broken, allowing programmers to comprehend and manage larger programs.

C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language “C with Classes.” However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C’s features, attributes, and benefits. This is a crucial reason for the success of C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

The Stage Is Set for Java

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed,
for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few years, the World Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming.

The Creation of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff,
Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of
CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (platform-independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems. Further, because (at that time) much of the computer world had divided itself
into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance. After all, the Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs. Even though many kinds of platforms are attached to the Internet, users would like them all to be able to run the same program. What was once an irritating but low-priority problem had become a high-profile necessity.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while
the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java’s large-scale success.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer’s language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for
professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the “Internet version of C++.” However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

As mentioned at the start of this chapter, computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs.
destined for distribution on the Internet. However, Java also embodies changes in the way that people approach the writing of programs. For example, Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access. In the final analysis, though, it was not the individual features of Java that made it so remarkable. Rather, it was the language as a whole. Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe. Java was to Internet programming what C was to system programming: a revolutionary force that changed the world.

The C# Connection

The reach and power of Java continues to be felt in the world of computer language development. Many of its innovative features, constructs, and concepts have become part of the baseline for any new language. The success of Java is simply too important to ignore.

Perhaps the most important example of Java’s
influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java. For example, both share the same general syntax, support distributed programming, and utilize the same object model. There are, of course, differences between Java and C#, but the overall “look and feel” of these languages is very similar. This “cross-pollination” from Java to C# is the strongest testimonial to date that Java redefined the way we think about and use a computer language.

How Java Changed the Internet
The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let’s look more closely at each of these.
Java Applets

An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For
example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program’s code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems. As you will see, Java solved these problems in an effective and elegant way. Let’s look a bit more closely at each.

Security

As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem
is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run
on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The same code must work on all computers. Therefore, some means of generating portable executable code was needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

Java’s Magic: The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). In essence, the original JVM was designed as an
interpreter for bytecode. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs. Here is why.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.
The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java’s initial release. HotSpot provides a Just-In-Time (JIT) compiler
for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

Servlets: Java on the Server Side
As useful as applets can be, they are just one half of the client/server equation. Not long after the initial release of Java, it became obvious that Java would also be
useful on the server side. The result was the servlet. A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet, Java spanned both sides of the client/server connection.

Servlets are used to create dynamically generated content that is then served to the client. For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser. Although dynamically generated content is available through mechanisms such as CGI (Common Gateway Interface), the servlet offers several advantages, including increased performance.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements
are that the server support the JVM and a servlet container.

The Java Buzzwords

No discussion of Java’s history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies.

Simple
Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented
Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist’s “everything is an object” paradigm and the pragmatist’s “stay out of my way” model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the
same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by
managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java’s easy-to-use approach to multithreading allows you to think about the specific
behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java’s creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, any time, forever.” To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be
executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and
resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java’s era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted
from attributes of its original specification.

The next major release of Java was Java 2, where the “2” indicates “second generation.” The creation of Java 2 was a watershed event, marking the beginning of Java’s “modern age.” The first release of Java 2 carried the version number 1.2. It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the Thread class in which the methods `suspend()`, `resume()`, and `stop()` were deprecated.
J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and “tightened up” the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword assert, chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java
upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language. To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

- Generics
- Annotations
- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style for loop
- Variable-length arguments (varargs)
- Static import
- Formatted I/O
- Concurrency utilities

This is not a list of minor tweaks or incremental upgrades. Each item in the list represented a significant addition to the Java language. Some, such as generics, the enhanced for, and varargs, introduced new syntax
elements. Others, such as autoboxing and auto-unboxing, altered the semantics of the language. Annotations added an entirely new dimension to programming. In all cases, the impact of these additions went beyond their direct effects. They changed the very character of Java itself.

The importance of these new features is reflected in the use of the version number “5.” The next version number for Java would normally have been 1.5. However, the new features were so significant that a shift from 1.4 to 1.5 just didn’t seem to express the magnitude of the change. Instead, Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the Developer’s Kit was called JDK 5. However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number, which is also referred to as the developer version number. The “5” in J2SE 5 is called the product version number.

The next release of Java was called Java SE 6. Sun once again decided to change the name of the Java
platform. First, notice that the “2” was dropped. Thus, the platform was now named Java SE, and the official product name was Java Platform, Standard Edition 6. The Java Developer’s Kit was called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.

Java SE 6 built on the base of J2SE 5, adding incremental improvements. Java SE 6 added no major features to the Java language proper, but it did enhance the API libraries, added several new packages, and offered improvements to the runtime. It also went through several updates during its (in Java terms) long life cycle, with several upgrades added along the way. In general, Java SE 6 served to further solidify the advances made by J2SE 5.

Java SE 7

The newest release of Java is called Java SE 7, with the Java Developer’s Kit being called JDK 7, and an internal version number of 1.7. Java SE 7 is the first major release of Java since Sun Microsystems was acquired by
Oracle (a process that began in April 2009 and that was completed in January 2010). Java SE 7 contains many new features, including significant additions to the language and the API libraries. Upgrades to the Java run-time system that support non-Java languages are also included, but it is the language and library additions that are of most interest to Java programmers.

The new language features were developed as part of Project Coin. The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7. Although these new features are collectively referred to as “small,” the effects of these changes are quite large in terms of the code they impact. In fact, for many programmers, these changes may well be the most important new features in Java SE 7. Here is a list of the new language features:

• A String can now control a switch statement.

• Binary integer literals.

• Underscores in numeric literals.

• An expanded try statement, called try-with-resources,
that supports automatic resource management. (For example, streams can now be closed automatically when they are no longer needed.)

- Type inference (via the diamond operator) when constructing a generic instance.
- Enhanced exception handling in which two or more exceptions can be caught by a single catch (multi-catch) and better type checking for exceptions that are rethrown.
- Although not a syntax change, the compiler warnings associated with some types of varargs methods have been improved, and you have more control over the warnings.

As you can see, even though the Project Coin features were considered small changes to the language, their benefits will be much larger than the qualifier “small” would suggest. In particular, the try-with-resources statement will profoundly affect the way that stream-based code is written. Also, the ability to now use a String to control a switch statement is a long-desired
improvement that will simplify coding in many situations.

Java SE 7 makes several additions to the Java API library. Two of the most important are the enhancements to the NIO Framework and the addition of the Fork/Join Framework. NIO (which originally stood for New I/O) was added to Java in version 1.4. However, the changes proposed for Java SE 7 fundamentally expand its capabilities. So significant are the changes, that the term NIO.2 is often used.

The Fork/Join Framework provides important support for parallel programming. Parallel programming is the name commonly given to the techniques that make effective use of computers that contain more than one processor, including multicore systems. The advantage that multicore environments offer is the prospect of significantly increased program performance. The Fork/Join Framework addresses parallel programming by

- Simplifying the creation and use of tasks that can
execute concurrently

- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can easily create scaleable applications that automatically take advantage of the processors available in the execution environment. Of course, not all algorithms lend themselves to parallelization, but for those that do, a significant improvement in execution speed can be obtained.

The material in this book has been updated to reflect Java SE 7, with many new features, updates, and additions indicated throughout.

A Culture of Innovation
Since the beginning, Java has been at the center of a culture of innovation. Its original release redefined programming for the Internet. The Java Virtual Machine (JVM) and bytecode changed the way we think about security and portability. The applet (and then the servlet) made the Web come alive. The Java Community
Process (JCP) redefined the way that new ideas are assimilated into the language. Because Java is used for Android programming, Java is part of the smartphone revolution. The world of Java has never stood still for very long. Java SE 7 is the latest release in Java’s ongoing, dynamic history.
As in all other computer languages, the elements of Java do not exist in isolation. Rather, they work together to form the language as a whole. However, this interrelatedness can make it difficult to describe one aspect of Java without involving several others. Often a discussion of one feature implies prior knowledge of another. For this reason, this chapter presents a quick overview of several key features of Java. The material described here will give you a foothold that will allow you to write and understand simple programs. Most of the topics discussed will be examined in greater detail in the remaining chapters of Part I.

Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is,
some programs are written around “what is happening” and others are written around “who is being affected.” These are the two paradigms that govern how a program is constructed. The first way is called the process-oriented model. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. However, as mentioned in Chapter 1, problems with this approach appear as programs grow larger and more complex.

To manage increasing complexity, the second approach, called object-oriented programming, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Abstraction
An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to
drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to do something. This is the essence of
Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism. Let’s take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled.
through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can’t affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver’s point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java, the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A class defines the structure and behavior (data and code) that will be shared by a set of objects.
Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods. (If you are familiar with C/C++, it may help to know that what a Java programmer calls a method, a C/C++ programmer calls a function.) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class.
Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class’ public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let’s take a closer look at this process.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical
way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the class definition for animals.

Figure 2-1 Encapsulation: public methods can be used to protect private data.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth and mammary
glands. This is known as a subclass of animals, where animals are referred to as mammals’ superclass.

Since mammals are simply more precisely specified animals, they inherit all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization (see Figure 2-2). This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.
Figure 2-2 Labrador inherits the encapsulation of all its superclasses.
Polymorphism

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler’s job to select the specific action (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

Extending the dog analogy, a dog’s sense of smell is
polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog’s nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

Polymorphism, Encapsulation, and Inheritance Work Together
When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scaleable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code.

Of the two real-world examples, the automobile more completely illustrates the power of object-oriented design. Dogs are fun to think about from an inheritance standpoint, but cars are more like programs. All drivers rely on inheritance to drive different types (subclasses) of vehicles. Whether the vehicle is a school bus, a Mercedes sedan, a Porsche, or the family minivan, drivers can all
more or less find and operate the steering wheel, the brakes, and the accelerator. After a bit of gear grinding, most people can even manage the difference between a stick shift and an automatic, because they fundamentally understand their common superclass, the transmission.

People interface with encapsulated features on cars all the time. The brake and gas pedals hide an incredible array of complexity with an interface so simple you can operate them with your feet! The implementation of the engine, the style of brakes, and the size of the tires have no effect on how you interface with the class definition of the pedals.

The final attribute, polymorphism, is clearly reflected in the ability of car manufacturers to offer a wide array of options on basically the same vehicle. For example, you can get an antilock braking system or traditional brakes, power or rack-and-pinion steering, and 4-, 6-, or 8-cylinder engines. Either way, you will still press the brake pedal to stop, turn the steering wheel to change direction, and press the accelerator when you want to move. The same interface can be used to control a number of different implementations.

As you can see, it is through the application of encapsulation, inheritance, and polymorphism that the individual parts are transformed into the object known as a car. The same is also true of computer programs. By the application of object-oriented principles, the various
parts of a complex program can be brought together to form a cohesive, robust, maintainable whole.

As mentioned at the start of this section, every Java program is object-oriented. Or, put more precisely, every Java program involves encapsulation, inheritance, and polymorphism. Although the short example programs shown in the rest of this chapter and in the next few chapters may not seem to exhibit all of these features, they are nevertheless present. As you will see, many of the features supplied by Java are part of its built-in class libraries, which do make extensive use of encapsulation, inheritance, and polymorphism.

A First Simple Program
Now that the basic object-oriented underpinning of Java has been discussed, let’s look at some actual Java programs. Let’s start by compiling and running the short sample program shown here. As you will see, this involves a little more work than you might imagine.

```java
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
class Example {
   // Your program begins with a call to main().
   public static void main(String args[]) {
      System.out.println("This is a simple Java program.");
   }
}
```

NOTE
The descriptions that follow use the standard Java SE 7 Development Kit (JDK 7), which is available from Oracle. If you are using a different Java development environment, then you may need to follow a different procedure for compiling and executing Java programs. In this case, consult your compiler’s documentation for details.

Entering the Program

For most computer languages, the name of the file that holds the source code to a program is immaterial. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be Example.java. Let’s see why.

In Java, a source file is officially called a compilation unit. It is a text file that contains (among other things) one or more class definitions. (For now, we will be using source files that contain only one class.) The Java compiler requires that a source file use the .java filename extension.

As you can see by looking at the program, the name of the class defined by the program is also Example. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of the main class should match the name of the file that holds the program. You
should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the Program

To compile the Example program, execute the compiler, javac, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The javac compiler creates a file called Example.class that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of javac is not code that can be directly executed.

To actually run the program, you must use the Java application launcher called java. To do so, pass the class name Example as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```
When Java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the .class file. When you execute java as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the .class extension. If it finds the file, it will execute the code contained in the specified class.

A Closer Look at the First Sample Program

Although Example.java is quite short, it includes several key features that are common to all Java programs. Let’s closely examine each part of the program.

The program begins with the following lines:

```java
/*
   This is a simple Java program.
   Call this file "Example.java".
*/
```

This is a comment. Like most other programming languages, Java lets you enter a remark into a program’s source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be
called Example.java. Of course, in real applications, comments generally explain how some part of the program works or what a specific feature does.

Java supports three styles of comments. The one shown at the top of the program is called a multiline comment. This type of comment must begin with /* and end with */. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown here:

class Example {

This line uses the keyword class to declare that a new class is being defined. Example is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). For the moment, don’t worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented.

The next line in the program is the single-line comment, shown here:

// Your program begins with a call to main().

This is the second type of comment supported by Java. A single-line comment begins with a // and ends at the end of the line. As a general rule, programmers use
multiline comments for longer remarks and single-line comments for brief, line-by-line descriptions. The third type of comment, a documentation comment, will be discussed in the “Comments” section later in this chapter.

The next line of code is shown here:

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

This line begins the `main( )` method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling `main( )`. The full meaning of each part of this line cannot be given now, since it involves a detailed understanding of Java’s approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let’s take a brief look at each part now.

The `public` keyword is an access modifier, which allows the programmer to control the visibility of class members. When a class member is preceded by `public`, then that member may be accessed by code outside the class in which it is declared. (The opposite of `public` is `private`, which prevents a member from being used by code defined outside of its class.) In this case, `main( )` must be declared as `public`, since it must be called by code outside of its class when the program is started. The keyword `static` allows `main( )` to be called without having to instantiate a particular instance of the class.
This is necessary since main() is called by the Java Virtual Machine before any objects are made. The keyword void simply tells the compiler that main() does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don’t worry. All of these concepts will be discussed in detail in subsequent chapters.

As stated, main() is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, Main is different from main. It is important to understand that the Java compiler will compile classes that do not contain a main() method. But java has no way to run these classes. So, if you had typed Main instead of main, the compiler would still compile your program. However, java would report an error because it would be unable to find the main() method.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. If there are no parameters required for a given method, you still need to include the empty parentheses. In main(), there is only one parameter, albeit a complicated one. String args[] declares a parameter named args, which is an array of instances of the class String. (Arrays are collections of similar objects.) Objects of type String store character
strings. In this case, args receives any command-line arguments present when the program is executed. This program does not make use of this information, but other programs shown later in this book will.

The last character on the line is the {. This signals the start of main( )’s body. All of the code that comprises a method will occur between the method’s opening curly brace and its closing curly brace.

One other point: main( ) is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a main( ) method to get things started. Furthermore, in some cases, you won’t need main( ) at all. For example, when creating applets—Java programs that are embedded in web browsers—you won’t use main( ) since the web browser uses a different means of starting the execution of applets.

The next line of code is shown here. Notice that it occurs inside main( ).

```java
System.out.println("This is a simple Java program.");
```

This line outputs the string “This is a simple Java program.” followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. In this case, println( ) displays the string which is passed to it. As you will see, println( ) can be used to display other types of information, too. The line begins
with System.out. While too complicated to explain in
detail at this time, briefly, System is a predefined class
that provides access to the system, and out is the output
stream that is connected to the console.

As you have probably guessed, console output (and
input) is not used frequently in most real-world Java
applications. Since most modern computing
environments are windowed and graphical in nature,
console I/O is used mostly for simple utility programs,
demonstration programs, and server-side code. Later in
this book, you will learn other ways to generate output
using Java. But for now, we will continue to use the
console I/O methods.

Notice that the println( ) statement ends with a
semicolon. All statements in Java end with a semicolon.
The reason that the other lines in the program do not
end in a semicolon is that they are not, technically,
statements.

The first } in the program ends main( ), and the last }
ends the Example class definition.

A Second Short Program
Perhaps no other concept is more fundamental to a
programming language than that of a variable. As you
probably know, a variable is a named memory location
that may be assigned a value by your program. The value
of a variable may be changed during the execution of the
program. The next program shows how a variable is
declared and how it is assigned a value. The program also illustrates some new aspects of console output. As the comments at the top of the program state, you should call this file Example2.java.

```java
class Example2 {
    public static void main(String args []) {
        int num; // this declares a variable called num
        num = 100; // this assigns num the value 100
        System.out.println("This is num: "+ num);
        num = num * 2;
        System.out.print("The value of num * 2 is ");
        System.out.println(num);
    }
}
```

When you run this program, you will see the following output:

```
This is num: 100
The value of num * 2 is 200
```

Let's take a close look at why this output is generated.

The first new line in the program is shown here:

```java
int num; // this declares a variable called num
```

This line declares an integer variable called num. Java (like most other languages) requires that variables be
Following is the general form of a variable declaration:

    type var-name;

Here, type specifies the type of variable being declared, and var-name is the name of the variable. If you want to declare more than one variable of the specified type, you may use a comma-separated list of variable names. Java defines several data types, including integer, character, and floating-point. The keyword int specifies an integer type.

In the program, the line

    num = 100; // this assigns num the value 100

assigns to num the value 100. In Java, the assignment operator is a single equal sign.

The next line of code outputs the value of num preceded by the string “This is num:”.

    System.out.println("This is num: " + num);

In this statement, the plus sign causes the value of num to be appended to the string that precedes it, and then the resulting string is output. (Actually, num is first converted from an integer into its string equivalent and then concatenated with the string that precedes it. This process is described in detail later in this book.)
approach can be generalized. Using the + operator, you can join together as many items as you want within a single println() statement.

The next line of code assigns num the value of num times 2. Like most other languages, Java uses the * operator to indicate multiplication. After this line executes, num will contain the value 200.

Here are the next two lines in the program:

```java
System.out.print("The value of num * 2 is ");
System.out.println(num);
```

Several new things are occurring here. First, the built-in method print() is used to display the string “The value of num * 2 is“. This string is not followed by a newline. This means that when the next output is generated, it will start on the same line. The print() method is just like println(), except that it does not output a newline character after each call. Now look at the call to println(). Notice that num is used by itself. Both print() and println() can be used to output values of any of Java’s built-in types.

Two Control Statements
Although Chapter 5 will look closely at control statements, two are briefly introduced here so that they can be used in example programs in Chapters 3 and 4. They will also help illustrate an important aspect of Java: blocks of code.
The if Statement
The Java if statement works much like the IF statement in any other language. Further, it is syntactically identical to the if statements in C, C++, and C#. Its simplest form is shown here:

```java
if (condition) statement;
```

Here, condition is a Boolean expression. If condition is true, then the statement is executed. If condition is false, then the statement is bypassed. Here is an example:

```java
if(num < 100) System.out.println("num is less than 100");
```

In this case, if num contains a value that is less than 100, the conditional expression is true, and println( ) will execute. If num contains a value greater than or equal to 100, then the println( ) method is bypassed.

As you will see in Chapter 4, Java defines a full complement of relational operators which may be used in a conditional expression. Here are a few:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Equal to</td>
</tr>
</tbody>
</table>

Notice that the test for equality is the double equal sign.

Here is a program that illustrates the if statement:

```java
/*
 * Demonstrate the if.
*/
```
Call this file "IfSample.java".
*/
class IfSample {
    public static void main(String args[]) {
        int x, y;

        x = 10;
y = 20;

        if(x < y) System.out.println("x is less than y");
x = x * 2;
        if(x == y) System.out.println("x now equal to y");
x = x * 2;
        if(x > y) System.out.println("x now greater than y");
// this won't display anything
        if(x == y) System.out.println("you won't see this");
    }
}

The output generated by this program is shown here:

x is less than y
x now equal to y
x now greater than y

Notice one other thing in this program. The line

int x, y;

declares two variables, x and y, by use of a comma-separated list.

The for Loop
As you may know from your previous programming experience, loop statements are an important part of nearly any programming language. Java is no exception.
In fact, as you will see in Chapter 5, Java supplies a powerful assortment of loop constructs. Perhaps the most versatile is the for loop. The simplest form of the for loop is shown here:

```
for (initialization; condition; iteration) statement;
```

In its most common form, the initialization portion of the loop sets a loop control variable to an initial value. The condition is a Boolean expression that tests the loop control variable. If the outcome of that test is true, the for loop continues to iterate. If it is false, the loop terminates. The iteration expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the for loop:

```
/*
   Demonstrate the for loop.

   Call this file "ForTest.java".
*/
class ForTest {
   public static void main(String args[]) {
      int x;

      for(x = 0; x<10; x = x+1)
         System.out.println("This is x: " + x);
   }
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
```
In this example, x is the loop control variable. It is initialized to zero in the initialization portion of the for. At the start of each iteration (including the first one), the conditional test \( x < 10 \) is performed. If the outcome of this test is true, the println( ) statement is executed, and then the iteration portion of the loop is executed. This process continues until the conditional test is false.

As a point of interest, in professionally written Java programs you will almost never see the iteration portion of the loop written as shown in the preceding program. That is, you will seldom see statements like this:

\[
x = x + 1;
\]

The reason is that Java includes a special increment operator which performs this operation more efficiently. The increment operator is ++. (That is, two plus signs back to back.) The increment operator increases its operand by one. By use of the increment operator, the preceding statement can be written like this:

\[
x++;
\]

Thus, the for in the preceding program will usually be written like this:
You might want to try this. As you will see, the loop still runs exactly the same as it did before.

Java also provides a decrement operator, which is specified as `- -`. This operator decreases its operand by one.

Using Blocks of Code
Java allows two or more statements to be grouped into blocks of code, also called code blocks. This is done by enclosing the statements between opening and closing curly braces. Once a block of code has been created, it becomes a logical unit that can be used any place that a single statement can. For example, a block can be a target for Java's `if` and `for` statements. Consider this `if` statement:

```java
if(x < y) { // begin a block
  x = y;
  y = 0;
} // end of block
```

Here, if `x` is less than `y`, then both statements inside the block will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block.

Let's look at another example. The following program uses a block of code as the target of a `for` loop.
Demonstrate a block of code.

Call this file "BlockTest.java"

```java
class BlockTest {
    public static void main(String args[]) {
        int x, y;

        y = 20;

        // the target of this loop is a block
        for(x = 0; x<10; x++) {
            System.out.println("This is x: "+x);
            System.out.println("This is y: "+y);
            y = y - 2;
        }
    }
}
```

The output generated by this program is shown here:

This is x: 0
This is y: 20
This is x: 1
This is y: 18
This is x: 2
This is y: 16
This is x: 3
This is y: 14
This is x: 4
This is y: 12
This is x: 5
This is y: 10
This is x: 6
This is y: 8
This is x: 7
This is y: 6
This is x: 8
This is y: 4
This is x: 9
This is y: 2
In this case, the target of the for loop is a block of code and not just a single statement. Thus, each time the loop iterates, the three statements inside the block will be executed. This fact is, of course, evidenced by the output generated by the program.

As you will see later in this book, blocks of code have additional properties and uses. However, the main reason for their existence is to create logically inseparable units of code.

Lexical Issues
Now that you have seen several short Java programs, it is time to more formally describe the atomic elements of Java. Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords. The operators are described in the next chapter. The others are described next.

Whitespace
Java is a free-form language. This means that you do not need to follow any special indentation rules. For instance, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

Identifiers
Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so \texttt{VALUE} is a different identifier than \texttt{Value}.

Some examples of valid identifiers are

<table>
<thead>
<tr>
<th>AvgTemp</th>
<th>count</th>
<th>a4</th>
<th>$test</th>
<th>this_is_ok</th>
</tr>
</thead>
</table>

Invalid identifier names include these:

<table>
<thead>
<tr>
<th>2count</th>
<th>high-temp</th>
<th>Not/ok</th>
</tr>
</thead>
</table>

Literals

A constant value in Java is created by using a literal representation of it. For example, here are some literals:

<table>
<thead>
<tr>
<th>100</th>
<th>98.6</th>
<th>‘X’</th>
<th>“This is a test”</th>
</tr>
</thead>
</table>

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a documentation comment. This type of comment is used to produce an
HTML file that documents your program. The documentation comment begins with a /** and ends with a */. Documentation comments are explained in the Appendix.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parentheses</td>
<td>Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.</td>
</tr>
<tr>
<td>{ }</td>
<td>Braces</td>
<td>Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets</td>
<td>Used to declare array types. Also used when dereferencing array values.</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
<td>Terminates statements.</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
<td>Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.</td>
</tr>
<tr>
<td>.</td>
<td>Period</td>
<td>Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.</td>
</tr>
</tbody>
</table>

The Java Keywords

There are 50 keywords currently defined in the Java language (see Table 2-1). These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as identifiers. Thus, they cannot be used as
names for a variable, class, or method.

The keywords `const` and `goto` are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use. However, the current specification for Java defines only the keywords shown in Table 2-1.

<table>
<thead>
<tr>
<th>abstract</th>
<th>continue</th>
<th>for</th>
<th>new</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>default</td>
<td>goto</td>
<td>package</td>
<td>synchronized</td>
</tr>
<tr>
<td>boolean</td>
<td>do</td>
<td>if</td>
<td>private</td>
<td>this</td>
</tr>
<tr>
<td>break</td>
<td>double</td>
<td>implements</td>
<td>protected</td>
<td>throw</td>
</tr>
<tr>
<td>byte</td>
<td>else</td>
<td>import</td>
<td>public</td>
<td>throws</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>instanceof</td>
<td>return</td>
<td>transient</td>
</tr>
<tr>
<td>catch</td>
<td>extends</td>
<td>int</td>
<td>short</td>
<td>try</td>
</tr>
<tr>
<td>char</td>
<td>final</td>
<td>interface</td>
<td>static</td>
<td>void</td>
</tr>
<tr>
<td>class</td>
<td>finally</td>
<td>long</td>
<td>strictfp</td>
<td>volatile</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>native</td>
<td>super</td>
<td>while</td>
</tr>
</tbody>
</table>

Table 2-1 Java Keywords

In addition to the keywords, Java reserves the following: `true`, `false`, and `null`. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

The Java Class Libraries

The sample programs shown in this chapter make use of two of Java’s built-in methods: `println()` and `print()`. As mentioned, these methods are members of the `System` class, which is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in
class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for windowed output. Thus, Java as a totality is a combination of the Java language itself, plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, part of becoming a Java programmer is learning to use the standard Java classes. Throughout Part I of this book, various elements of the standard library classes and methods are described as needed. In Part II, the class libraries are described in detail.
CHAPTER 3
Data Types, Variables, and Arrays

This chapter examines three of Java’s most fundamental elements: data types, variables, and arrays. As with all modern programming languages, Java supports several types of data. You may use these types to declare variables and to create arrays. As you will see, Java’s approach to these items is clean, efficient, and cohesive.

Java Is a Strongly Typed Language

It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java’s safety and robustness comes from this fact. Let’s see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

The Primitive Types

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types, and
both terms will be used in this book. These can be put in four groups:

- **Integers** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers** This group includes float and double, which represent numbers with fractional precision.
- **Characters** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean** This group includes boolean, which is a special type for representing true/false values.

You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

The primitive types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non-object-oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.

The primitive types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment.
However, Java is different. Because of Java’s portability requirement, all data types have a strictly defined range. For example, an int is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability.

Let’s look at each type of data in turn.

Integers
Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers. However, Java’s designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of unsigned was used mostly to specify the behavior of the high-order bit, which defines the sign of an integer value. As you will see in Chapter 4, Java manages the meaning of the high-order bit differently, by adding a special “unsigned right shift” operator. Thus, the need for an unsigned integer type was eliminated.

The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that
type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>64</td>
<td>(-9,223,372,036,854,775,808) to (9,223,372,036,854,775,807)</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>(-2,147,483,648) to (2,147,483,647)</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>(-32,768) to (32,767)</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>(-128) to (127)</td>
</tr>
</tbody>
</table>

Let's look at each type of integer.

byte

The smallest integer type is byte. This is a signed 8-bit type that has a range from \(-128\) to \(127\). Variables of type byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.

Byte variables are declared by use of the byte keyword. For example, the following declares two byte variables called b and c:

```java
byte b, c;
```

short

short is a signed 16-bit type. It has a range from \(-32,768\) to \(32,767\). It is probably the least-used Java type. Here are some examples of short variable declarations:
int

The most commonly used integer type is int. It is a signed 32-bit type that has a range from \(-2,147,483,648\) to \(2,147,483,647\). In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case. The reason is that when byte and short values are used in an expression they are promoted to int when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, int is often the best choice when an integer is needed.

long

long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days:

```java
// Compute distance light travels using long variables.
class Light {
    public static void main(String args[]) {
        int lightspeed;
```
long days;
long seconds;
long distance;

// approximate speed of light in miles per second
lightspeed = 186000;

days = 1000; // specify number of days here
seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance

System.out.print("In " + days);
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}

This program generates the following output:

In 1000 days light will travel about 16070400000000 miles.

Clearly, the result could not have been held in an int variable.

Floating-Point Types
Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:
Each of these floating-point types is examined next.

float

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don’t require a large degree of precision. For example, float can be useful when representing dollars and cents.

Here are some example float variable declarations:

```c
float hightemp, lowtemp;
```

double

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

Here is a short program that uses double variables to
// Compute the area of a circle.

class Area {
    public static void main(String args[]) {
        double pi, r, a;

        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately

        a = pi * r * r; // compute area

        System.out.println("Area of circle is " + a);
    }
}

Character

In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters.
Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.

NOTE

More information about Unicode can be found at http://www.unicode.org.

Here is a program that demonstrates char variables:

```java
// Demonstrate char data type.
class CharDemo {
    public static void main(String args[]) {
        char ch1, ch2;

        ch1 = 88; // code for X
        ch2 = 'Y';

        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the “old tricks” that you may have used with characters in other languages will work in Java, too.
Although char is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable. Consider the following program:

```
// char variables behave like integers.
class CharDemo2 {
    public static void main(String args[]) {
        char ch1;
        ch1 = 'X';
        System.out.println("ch1 contains " + ch1);
        ch1++; // increment ch1
        System.out.println("ch1 is now " + ch1);
    }
}
```

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

In the program, ch1 is first given the value X. Next, ch1 is incremented. This results in ch1 containing Y, the next character in the ASCII (and Unicode) sequence.

**NOTE**

In the formal specification for Java, char is referred to as an integral type, which means that it is in the same general category as int, short, long, and byte. However, because its principal use is for representing Unicode characters, char is commonly considered to be in a category of its own.
Booleans
Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the case of \( a < b \). boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

Here is a program that demonstrates the boolean type:

```java
// Demonstrate boolean values.
class BoolTest {
    public static void main(String args[]) {
        boolean b;

        b = false;
        System.out.println("b is "+b);
        b = true;
        System.out.println("b is "+b);

        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");

        b = false;
        if(b) System.out.println("This is not executed.");

        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is "+(10 > 9));
    }
}
```

The output generated by this program is shown here:

b is false
b is true
This is executed.
10 > 9 is true

There are three interesting things to notice about this program. First, as you can see, when a boolean value is
output by println( ), “true” or “false” is displayed.
Second, the value of a boolean variable is sufficient, by itself, to control the if statement. There is no need to write an if statement like this:

    if(b == true) ...

Third, the outcome of a relational operator, such as <, is a boolean value. This is why the expression 10 > 9 displays the value “true.” Further, the extra set of parentheses around 10 > 9 is necessary because the + operator has a higher precedence than the >.

A Closer Look at Literals

Literals were mentioned briefly in Chapter 2. Now that the built-in types have been formally described, let’s take a closer look at them.

Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. There are two other bases which can be used in integer literals, octal (base eight) and hexadecimal (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal’s 0 to 7 range. A more common base for numbers used by
programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits. You signify a hexadecimal constant with a leading zero-x, (0x or 0X). The range of a hexadecimal digit is 0 to 15, so A through F (or a through f) are substituted for 10 through 15.

Integer literals create an int value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java’s other integer types, such as byte or long, without causing a type mismatch error. Fortunately, such situations are easily handled. When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a long variable. However, to specify a long literal, you will need to explicitly tell the compiler that the literal value is of type long. You do this by appending an upper- or lowercase L to the literal. For example, 0x7fffffffL or 9223372036854775807L is the largest long. An integer can also be assigned to a char as long as it is within range.

Beginning with JDK 7, you can also specify integer literals using binary. To do so, prefix the value with 0b or 0B. For example, this specifies the decimal value 10 using a binary literal:

    int x = 0b1010;
Among other uses, the addition of binary literals makes it easier to enter values used as bitmasks. In such a case, the decimal (or hexadecimal) representation of the value does not visually convey its meaning relative to its use. The binary literal does.

Also beginning with JDK 7, you can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given

```java
int x = 123_456_789;
```

the value given to `x` will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. For example, this is valid:

```java
int x = 123___456___789;
```

The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on. They are also useful for providing visual groupings when specifying binary literals. For example, binary values are often visually grouped in four-digits units, as shown here:

```java
int x = 0b1101_0101_0001_1010;
```

Floating-Point Literals
Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. Standard notation consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers. Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.

Floating-point literals in Java default to double precision. To specify a float literal, you must append an F or f to the constant. You can also explicitly specify a double literal by appending a D or d. Doing so is, of course, redundant. The default double type consumes 64 bits of storage, while the smaller float type requires only 32 bits.

Hexadecimal floating-point literals are also supported, but they are rarely used. They must be in a form similar to scientific notation, but a P or p, rather than an E or e, is used. For example, 0x12.2P2 is a valid floating-point literal. The value following the P, called the binary exponent, indicates the power-of-two by which the number is multiplied. Therefore, 0x12.2P2 represents
Beginning with JDK 7, you can embed one or more underscores in a floating-point literal. This feature works the same as it does for integer literals, which were just described. Its purpose is to make it easier to read large floating-point literals. When the literal is compiled, the underscores are discarded. For example, given

```java
double num = 9_423_497_862.0;
```

the value given to num will be 9,423,497,862.0. The underscores will be ignored. As is the case with integer literals, underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. It is also permissible to use underscores in the fractional portion of the number. For example,

```java
double num = 9_423_497.1_0_9;
```

is legal. In this case, the fractional part is .109.

**Boolean Literals**

Boolean literals are simple. There are only two logical values that a boolean value can have, true and false. The values of true and false do not convert into any numerical representation. The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, the Boolean literals can only be assigned to variables declared as boolean or used in expressions with Boolean
operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as ‘a’, ‘z’, and ‘@’. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as ‘\’ for the single-quote character itself and ‘\n’ for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, ‘\141’ is the letter ‘a’. For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits. For example, ‘\u0061’ is the ISO-Latin-1 ‘a’ because the top byte is zero. ‘\ua432’ is a Japanese Katakana character. Table 3-1 shows the character escape sequences.

String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are
“Hello World”
“two\nlines”
“\"This is in quotes\"”

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\didd</td>
<td>Octal character (ddd)</td>
</tr>
<tr>
<td>\uxxxx</td>
<td>Hexadecimal Unicode character (xxxx)</td>
</tr>
<tr>
<td>\</td>
<td>Single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\n</td>
<td>New line (also known as line feed)</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
</tbody>
</table>

Table 3-1 Character Escape Sequences

The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages.

NOTE
As you may know, in some other languages, including C/C++, strings are implemented as arrays of characters. However, this is not the case in Java. Strings are actually object types. As you will see later in this book, because Java implements strings as objects, Java includes extensive string-handling
variables that are both powerful and easy to use.

Variables
The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

Declaring a Variable
In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```java
    type identifier [ = value ][, identifier [ = value ] ...];
```

The type is one of Java’s atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```java
    int a, b, c;             // declares three ints, a, b, and c.
    int d = 3, e, f = 5;     // declares three more ints, initializing d and f.
```
byte z = 22;         // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x';       // the variable x has the value 'x'.

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Java allows any properly formed identifier to have any declared type.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```java
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;

        // c is dynamically initialized
        double c = Math.sqrt(a * a + b * b);

        System.out.println("Hypotenuse is " + c);
    }
}
```

Here, three local variables—\(a\), \(b\), and \(c\)—are declared. The first two, \(a\) and \(b\), are initialized by constants. However, \(c\) is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java’s built-in methods, `sqrt()`, which is a member of the `Math` class, to compute the
square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the `main()` method. However, Java allows variables to be declared within any block. As explained in Chapter 2, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Many other computer languages define two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java’s strict, object-oriented model. While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. Because of the differences, a discussion of class scope (and variables declared within it) is deferred until
when classes are described. For now, we will only examine the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method’s scope. Although this book will look more closely at parameters in Chapter 6, for the sake of this discussion, they work the same as any other method variable.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```java
// Demonstrate block scope.
class Scope {
    public static void main(String args[]) {
        int x; // known to all code within main
```

Chapter 6,
x = 10;
if (x == 10) { // start new scope
    int y = 20; // known only to this block

    // x and y both known here.
    System.out.println("x and y: "+x+" " + y);
    x = y * 2;
} // y = 100; // Error! y not known here

// x is still known here.
System.out.println("x is " + x);
}

As the comments indicate, the variable x is declared at the start of main( )’s scope and is accessible to all subsequent code within main( ). Within the if block, y is declared. Since a block defines a scope, y is only visible to other code within its block. This is why outside of its block, the line y = 100; is commented out. If you remove the leading comment symbol, a compile-time error will occur, because y is not visible outside of its block. Within the if block, x can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it. For example, this fragment is invalid because count cannot be used prior to its declaration:
count = 100; // oops! cannot use count before it is declared!
int count;

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program:

```java
class LifeTime {
    public static void main(String args[]) {
        int x;

        for(x = 0; x < 3; x++) {
            int y = -1; // y is initialized each time block is entered
            System.out.println("y is: "+ y); // this always prints -1
            y = 100;
            System.out.println("y is now: "+ y);
        }
    }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
```
As you can see, y is reinitialized to -1 each time the inner for loop is entered. Even though it is subsequently assigned the value 100, this value is lost.

One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
    public static void main(String args[]) {
        int bar = 1;
        { // creates a new scope
            int bar = 2; // Compile-time error - bar already defined!
        }
    }
}
```

Type Conversion and Casting
If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a
conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let’s look at both automatic type conversions and casting.

Java’s Automatic Conversions
When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types
Although the automatic type conversions are helpful,
they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

```
(target-type) value
```

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer’s value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte’s range.

```java
int a;
byte b;
// ...
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23
will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type’s range.

The following program demonstrates some type conversions that require casts:

```java
// Demonstrate casts.
class Conversion {
    public static void main(String args[]) {
        byte b;
        int i = 257;
        double d = 323.142;

        System.out.println("\nConversion of int to byte.");
        b = (byte) i;
        System.out.println("i and b " + i + " " + b);

        System.out.println("\nConversion of double to int.");
        i = (int) d;
        System.out.println("d and i " + d + " " + i);

        System.out.println("\nConversion of double to byte.");
        b = (byte) d;
        System.out.println("d and b " + d + " " + b);
    }
}
```

This program generates the following output:

Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67

Let’s look at each conversion. When the value 257 is cast
into a byte variable, the result is the remainder of the division of 257 by 256 (the range of a byte), which is 1 in this case. When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Automatic Type Promotion in Expressions
In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```java
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term a * b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression. This means that the subexpression a*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this
seemingly correct code causes a problem:

```java
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store $50 \times 2$, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result has also been promoted to int. Thus, the result of the expression is now of type int, which cannot be assigned to a byte without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```java
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions. They are as follows: First, all byte, short, and char values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands are double, the result is double.

The following program demonstrates how each value
class Promote {
    public static void main(String args[]) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);

        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("result = " + result);
    }
}

Let's look closely at the type promotions that occur in this line from the program:

double result = (f * b) + (i / c) - (d * s);

In the first subexpression, \( f \times b \), \( b \) is promoted to a float and the result of the subexpression is float. Next, in the subexpression \( i/c \), \( c \) is promoted to int, and the result is of type int. Then, in \( d \times s \), the value of \( s \) is promoted to double, and the type of the subexpression is double. Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

Arrays
An array is a group of like-typed variables that are
referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

NOTE

If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.

One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

```
    type var-name[ ];
```

Here, type declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold. For example, the following declares an array named month_days with the type “array of int”:

```
    int month_days[ ];
```

Although this declaration establishes the fact that
month_days is an array variable, no array actually exists. In fact, the value of month_days is set to null, which represents an array with no value. To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days. new is a special operator that allocates memory.

You will look more closely at new in a later chapter, but you need to use it now to allocate memory for arrays. The general form of new as it applies to one-dimensional arrays appears as follows:

array-var = new type [size];

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference types, which are described in a later chapter). This example allocates a 12-element array of integers and links them to month_days:

    month_days = new int[12];

After this statement executes, month_days will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.
Let’s review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. If the concept of dynamic allocation is unfamiliar to you, don’t worry. It will be described at length later in this book.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of `month_days`:

```java
month_days[1] = 28;
```

The next line displays the value stored at index 3:

```java
System.out.println(month_days[3]);
```

Putting together all the pieces, here is a program that creates an array of the number of days in each month:

```java
// Demonstrate a one-dimensional array.
class Array {
    public static void main(String args[]) {
        int month_days[];
        month_days = new int[12];
        month_days[0] = 31;
        month_days[1] = 28;
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
        month_days[6] = 31;
        month_days[7] = 31;
```
When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is `month_days[3]` or 30.

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```java
int month_days[] = new int[12];
```

This is the way that you will normally see it done in professionally written Java programs.

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use `new`. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```java
// An improved version of the previous program.
class AutoArray {
    public static void main(String args[]) {
        month_days[8] = 30;
        month_days[9] = 31;
        month_days[10] = 30;
        month_days[11] = 31;
        System.out.println("April has " + month_days[3] + " days.");
    }
}
```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 30, 31, 30, 31, 31 };  
System.out.println("April has " + month_days[3] + " days.");

When you run this program, you see the same output as that generated by the previous version.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. For example, the run-time system will check the value of each index into month_days to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

// Average an array of values.
class Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;

        for(i=0; i<5; i++)
            result = result + nums[i];
        System.out.println("Average is " + result / 5);
    }
}
Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD:

```java
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to `twoD`. Internally this matrix is implemented as an array of arrays of `int`. Conceptually, this array will look like the one shown in Figure 3-1.

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```java
// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][] = new int[4][5];
        int i, j, k = 0;

        for(i=0; i<4; i++)
            for(j=0; j<5; j++) {
                twoD[i][j] = k;
                k++;
            }

        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
                System.out.print(twoD[i][j] + " ");
        }
    }
}
System.out.println();
}
}
}

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of `twoD` when it is declared. It allocates the second dimension manually.

```java
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```
While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

```java
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[]) {
        int twoD[][] = new int[4][5];
        // Additional code...
    }
}
```
This program generates the following output:

0
1 2
3 4 5
6 7 8 9

The array created by this program looks like this:

```
[0] 0
[1] 0 [1] 1
```

The use of uneven (or irregular) multidimensional
arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension’s initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes. Also notice that you can use expressions as well as literal values inside of array initializers.

```java
// Initialize a two-dimensional array.
class Matrix {
    public static void main(String args[]) {
        double m[][] = {
            { 0*0, 1*0, 2*0, 3*0 },
            { 0*1, 1*1, 2*1, 3*1 },
            { 0*2, 1*2, 2*2, 3*2 },
            { 0*3, 1*3, 2*3, 3*3 }
        };
        int i, j;

        for(i=0; i<4; i++) {
            for(j=0; j<4; j++)
                System.out.print(m[i][j] + " ");
            System.out.println();
        }
    }
}
```

When you run this program, you will get the following
As you can see, each row in the array is initialized as specified in the initialization lists.

Let's look at one more example that uses a multidimensional array. The following program creates a 3 by 4 by 5, three-dimensional array. It then loads each element with the product of its indexes. Finally, it displays these products.

```java
// Demonstrate a three-dimensional array.
class ThreeDMatrix {
    public static void main(String args[]) {
        int threeD[][][] = new int[3][4][5];
        int i, j, k;

        for(i=0; i<3; i++)
            for(j=0; j<4; j++)
                for(k=0; k<5; k++)
                    threeD[i][j][k] = i * j * k;

        for(i=0; i<3; i++) {
            for(j=0; j<4; j++) {
                for(k=0; k<5; k++)
                    System.out.print(threeD[i][j][k] + " ");
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

This program generates the following output:

```
0 0 0 0 0
```
Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

    type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

    int a1[] = new int[3];
    int[] a2 = new int[3];

The following declarations are also equivalent:

    char twod1[][] = new char[3][4];
    char[][] twod2 = new char[3][4];

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

    int[] nums, nums2, nums3; // create three arrays

creates three array variables of type int. It is the same as
int nums[], nums2[], nums3[]; // create three arrays

The alternative declaration form is also useful when specifying an array as a return type for a method. Both forms are used in this book.

A Few Words About Strings
As you may have noticed, in the preceding discussion of data types and arrays there has been no mention of strings or a string data type. This is not because Java does not support such a type—it does. It is just that Java’s string type, called String, is not a primitive type. Nor is it simply an array of characters. Rather, String defines an object, and a full description of it requires an understanding of several object-related features. As such, it will be covered later in this book, after objects are described. However, so that you can use simple strings in example programs, the following brief introduction is in order.

The String type is used to declare string variables. You can also declare arrays of strings. A quoted string constant can be assigned to a String variable. A variable of type String can be assigned to another variable of type String. You can use an object of type String as an argument to println(). For example, consider the following fragment:

```java
String str = "this is a test";
System.out.println(str);
```
Here, str is an object of type String. It is assigned the string “this is a test”. This string is displayed by the println() statement.

As you will see later, String objects have many special features and attributes that make them quite powerful and easy to use. However, for the next few chapters, you will be using them only in their simplest form.

A Note to C/C++ Programmers About Pointers

If you are an experienced C/C++ programmer, then you know that these languages provide support for pointers. However, no mention of pointers has been made in this chapter. The reason for this is simple: Java does not support or allow pointers. (Or more properly, Java does not support pointers that can be accessed and/or modified by the programmer.) Java cannot allow pointers, because doing so would allow Java programs to breach the firewall between the Java execution environment and the host computer. (Remember, a pointer can be given any address in memory—even addresses that might be outside the Java run-time system.) Since C/C++ make extensive use of pointers, you might be thinking that their loss is a significant disadvantage to Java. However, this is not true. Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one.
CHAPTER 4
Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations. This chapter describes all of Java’s operators except for the type comparison operator instanceof, which is examined in Chapter 13.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition (also unary plus)</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction (also unary minus)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
</tr>
<tr>
<td>+=</td>
<td>Addition assignment</td>
</tr>
<tr>
<td>-=</td>
<td>Subtraction assignment</td>
</tr>
<tr>
<td>*=</td>
<td>Multiplication assignment</td>
</tr>
<tr>
<td>/=</td>
<td>Division assignment</td>
</tr>
<tr>
<td>%=</td>
<td>Modulus assignment</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
</tr>
</tbody>
</table>

The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types,
but you can use them on char types, since the char type in Java is, essentially, a subset of int.

The Basic Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types. The unary minus operator negates its single operand. The unary plus operator simply returns the value of its operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```java
// Demonstrate the basic arithmetic operators.
class BasicMath {
    public static void main(String args[])
    {
        // arithmetic using integers
        System.out.println("Integer Arithmetic");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = "+ a);
        System.out.println("b = "+ b);
        System.out.println("c = "+ c);
        System.out.println("d = "+ d);
        System.out.println("e = "+ e);
        
        // arithmetic using doubles
        System.out.println("\nFloating Point Arithmetic");
        double da = 1 + 1;
        double db = da * 3;
        double dc = db / 4;
        double dd = dc - a;
        double de = -dd;
    }
}
```
When you run this program, you will see the following output:

Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1

Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5

The Modulus Operator

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the %:

```java
// Demonstrate the % operator.
class Modulus {
    public static void main(String args[]) {
        int x = 42;
        double y = 42.25;

        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}
```
When you run this program, you will get the following output:

\[
\begin{align*}
x \mod 10 &= 2 \\
y \mod 10 &= 2.25
\end{align*}
\]

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming:

\[a = a + 4;\]

In Java, you can rewrite this statement as shown here:

\[a += 4;\]

This version uses the \[+=\] compound assignment operator. Both statements perform the same action: they increase the value of \[a\] by 4.

Here is another example,

\[a = a \% 2;\]

which can be expressed as

\[a %= 2;\]

In this case, the \[\%=\] obtains the remainder of \[a / 2\] and puts that result back into \[a\].

There are compound assignment operators for all of the arithmetic, binary operators. Thus, any statement of the form

\[\text{var} = \text{var} \text{ op expression};\]
can be rewritten as

    var op = expression;

The compound assignment operators provide two benefits. First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms. Second, they are implemented more efficiently by the Java runtime system than are their equivalent long forms. For these reasons, you will often see the compound assignment operators used in professionally written Java programs.

Here is a sample program that shows several op= assignments in action:

    // Demonstrate several assignment operators.  
    class OpEquals {  
        public static void main(String args[]) {  
            int a = 1;  
            int b = 2;  
            int c = 3;  

            a += 5;  
            b *= 4;  
            c += a * b;  
            c %= 6;  
            System.out.println("a = " + a);  
            System.out.println("b = " + b);  
            System.out.println("c = " + c);  
        }  
    }

The output of this program is shown here:

    a = 6  
    b = 8  
    c = 3
Increment and Decrement

The ++ and the -- are Java’s increment and decrement operators. They were introduced in Chapter 2. Here they will be discussed in detail. As you will see, they have some special properties that make them quite interesting. Let’s begin by reviewing precisely what the increment and decrement operators do.

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

```
x = x + 1;
```

can be rewritten like this by use of the increment operator:

```
x++;
```

Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown, and prefix form, where they precede the operand. In the foregoing examples, there is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, then a subtle, yet powerful, difference between these two forms
appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified. For example:

```java
x = 42;
y = ++x;
```

In this case, y is set to 43 as you would expect, because the increment occurs before x is assigned to y. Thus, the line `y = ++x;` is the equivalent of these two statements:

```java
x = x + 1;
y = x;
```

However, when written like this,

```java
x = 42;
y = x++;
```

the value of x is obtained before the increment operator is executed, so the value of y is 42. Of course, in both cases x is set to 43. Here, the line `y = x++;` is the equivalent of these two statements:

```java
y = x;
x = x + 1;
```

The following program demonstrates the increment operator.

```java
// Demonstrate ++.
class IncDec {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c;
        int d;
```
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}

The output of this program follows:

a = 2
b = 3
c = 4
d = 1

The Bitwise Operators
Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands. They are summarized in the following table:
Since the bitwise operators manipulate the bits within an integer, it is important to understand what effects such manipulations may have on a value. Specifically, it is useful to know how Java stores integer values and how it represents negative numbers. So, before continuing, let’s briefly review these two topics.

All of the integer types are represented by binary numbers of varying bit widths. For example, the byte value for 42 in binary is 00101010, where each position represents a power of two, starting with $2^0$ at the rightmost bit. The next bit position to the left would be $2^1$, or 2, continuing toward the left with $2^2$, or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of $2^1 + 2^3 + 2^5$, which is $2 + 8 + 32$.

All of the integer types (except char) are signed.
All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as two’s complement, which means that negative numbers are represented by inverting (changing 1’s to 0’s and vice versa) all of the bits in a value, then adding 1 to the result. For example, $-42$ is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or $-42$. To decode a negative number, first invert all of the bits, then add 1. For example, $-42$, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

The reason Java (and most other computer languages) uses two’s complement is easy to see when you consider the issue of zero crossing. Assuming a byte value, zero is represented by 00000000. In one’s complement, simply inverting all of the bits creates 11111111, which creates negative zero. The trouble is that negative zero is invalid in integer math. This problem is solved by using two’s complement to represent negative values. When using two’s complement, 1 is added to the complement, producing 100000000. This produces a 1 bit too far to the left to fit back into the byte value, resulting in the desired behavior, where $-0$ is the same as 0, and 11111111 is the encoding for $-1$. Although we used a byte value in the preceding example, the same basic principle applies to all of Java’s integer types.

Because Java uses two’s complement to store negative numbers—and because all integers are signed values in
Java—applying the bitwise operators can easily produce unexpected results. For example, turning on the high-order bit will cause the resulting value to be interpreted as a negative number, whether this is what you intended or not. To avoid unpleasant surprises, just remember that the high-order bit determines the sign of an integer no matter how that high-order bit gets set.

The Bitwise Logical Operators

The bitwise logical operators are & , | , ^ , and ~ . The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A &amp; B</th>
<th>A ^ B</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~ , inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010

becomes

11010101

after the NOT operator is applied.
The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

\[
\begin{align*}
00101010 & \quad 42 \\
& \& 00001111 & \quad 15 \\
\hline
00001010 & \quad 10
\end{align*}
\]

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

\[
\begin{align*}
00101010 & \quad 42 \\
| & 00001111 & \quad 15 \\
\hline
00101111 & \quad 47
\end{align*}
\]

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.
Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

```java
// Demonstrate the bitwise logical operators.
class BitLogic {
    public static void main(String args[]) {
        String binary[] = {
            "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
            "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
        };
        int a = 3; // 0 + 2 + 1 or 0011 in binary
        int b = 6; // 4 + 2 + 0 or 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;
        System.out.println("        a = " + binary[a]);
        System.out.println("        b = " + binary[b]);
        System.out.println("      a|b = " + binary[c]);
        System.out.println("      a&b = " + binary[d]);
        System.out.println("      a^b = " + binary[e]);
        System.out.println("~a&b|a&~b = " + binary[f]);
        System.out.println("       ~a = " + binary[g]);
    }
}
```

In this example, a and b have bit patterns that present all four possibilities for two binary digits: 0-0, 0-1, 1-0, and 1-1. You can see how the | and & operate on each bit by the results in c and d. The values assigned to e and f are the same and illustrate how the ^ works. The string array named binary holds the human-readable, binary representation of the numbers 0 through 15. In this
example, the array is indexed to show the binary representation of each result. The array is constructed such that the correct string representation of a binary value \( n \) is stored in \( \text{binary}[n] \). The value of \( \sim a \) is ANDed with \( 0x0f \) (0000 1111 in binary) in order to reduce its value to less than 16, so it can be printed by use of the binary array. Here is the output from this program:

\[
\begin{array}{c}
\text{a} = 0011 \\
\text{b} = 0110 \\
\text{a} \lor \text{b} = 0111 \\
\text{a} \land \text{b} = 0010 \\
\text{a} \oplus \text{b} = 0101 \\
\sim \text{a} \land \text{b} \lor \text{a} \land \sim \text{b} = 0101 \\
\sim \text{a} = 1100
\end{array}
\]

The Left Shift

The left shift operator, \( << \), shifts all of the bits in a value to the left a specified number of times. It has this general form:

\[
\text{value} \ll \text{num}
\]

Here, \( \text{num} \) specifies the number of positions to left-shift the value in \( \text{value} \). That is, the \( \ll \) moves all of the bits in the specified value to the left by the number of bit positions specified by \( \text{num} \). For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an int operand, bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63.

Java’s automatic type promotions produce unexpected
results when you are shifting byte and short values. As you know, byte and short values are promoted to int when an expression is evaluated. Furthermore, the result of such an expression is also an int. This means that the outcome of a left shift on a byte or short value will be an int, and the bits shifted left will not be lost until they shift past bit position 31. Furthermore, a negative byte or short value will be sign-extended when it is promoted to int. Thus, the high-order bits will be filled with 1’s. For these reasons, to perform a left shift on a byte or short implies that you must discard the high-order bytes of the int result. For example, if you left-shift a byte value, that value will first be promoted to int and then shifted. This means that you must discard the top three bytes of the result if what you want is the result of a shifted byte value. The easiest way to do this is to simply cast the result back into a byte. The following program demonstrates this concept:

```java
// Left shifting a byte value.
public static void main(String args[]) {
  byte a = 64, b;
  int i;
  i = a << 2;
  b = (byte) (a << 2);

  System.out.println("Original value of a: " + a);
  System.out.println("i and b: " + i + " " + b);
}
```

The output generated by this program is shown here:

```
Original value of a: 64
```
Since a is promoted to int for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in i containing the value 256 (1 0000 0000). However, the value in b contains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has been shifted out.

Since each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2. But you need to watch out. If you shift a 1 bit into the high-order position (bit 31 or 63), the value will become negative. The following program illustrates this point:

```java
// Left shifting as a quick way to multiply by 2.
class MultByTwo {
    public static void main(String args[]) {
        int i;
        int num = 0xFFFFFFFF;

        for(i=0; i<4; i++) {
            num = num << 1;
            System.out.println(num);
        }
    }
}
```

The program generates the following output:

```
536870908
1073741816
2147483632
-32
```

The starting value was carefully chosen so that after being shifted left 4 bit positions, it would produce −32. As you can see, when a 1 bit is shifted into bit 31, the
The Right Shift

The right shift operator, `>>`, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

```
value >> num
```

Here, `num` specifies the number of positions to right-shift the value in `value`. That is, the `>>` moves all of the bits in the specified value to the right the number of bit positions specified by `num`.

The following code fragment shifts the value 32 to the right by two positions, resulting in `a` being set to 8:

```c
int a = 32;
a = a >> 2; // a now contains 8
```

When a value has bits that are “shifted off,” those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low-order bits to be lost, resulting again in `a` being set to 8:

```c
int a = 35;
a = a >> 2; // a contains 8
```

Looking at the same operation in binary shows more clearly how this happens:

```
00100011   35
>> 2
00001000    8
```
Each time you shift a value to the right, it divides that value by two—and discards any remainder. You can take advantage of this for high-performance integer division by 2. Of course, you must be sure that you are not shifting any bits off the right end.

When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called sign extension and serves to preserve the sign of negative numbers when you shift them right. For example, \(-8 \gg 1\) is \(-4\), which, in binary, is

\[
\begin{align*}
11111000 & \quad -8 \\
\gg & \quad 1 \\
11111100 & \quad -4
\end{align*}
\]

It is interesting to note that if you shift \(-1\) right, the result always remains \(-1\), since sign extension keeps bringing in more ones in the high-order bits.

Sometimes it is not desirable to sign-extend values when you are shifting them to the right. For example, the following program converts a byte value to its hexadecimal string representation. Notice that the shifted value is masked by ANDing it with \(0x0f\) to discard any sign-extended bits so that the value can be used as an index into the array of hexadecimal characters.

```java
// Masking sign extension.
class HexByte {
    static public void main(String args[]) {
        char hex[] = {
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
        };
    }
}
```
byte b = (byte) 0xf1;

System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
}
}

Here is the output of this program:

b = 0xf1

The Unsigned Right Shift
As you have just seen, the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is undesirable. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift. To accomplish this, you will use Java’s unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the >>>.
Here, a is set to −1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

int a = -1;
a = a >>> 24;
Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111    −1 in binary as an int
>>>24
00000000 00000000 00000000 11111111    255 in binary as an int
```

The `>>>` operator is often not as useful as you might like, since it is only meaningful for 32- and 64-bit values. Remember, smaller values are automatically promoted to int in expressions. This means that sign-extension occurs and that the shift will take place on a 32-bit rather than on an 8- or 16-bit value. That is, one might expect an unsigned right shift on a byte value to zero-fill beginning at bit 7. But this is not the case, since it is a 32-bit value that is actually being shifted. The following program demonstrates this effect:

```java
// Unsigned shifting a byte value.

class ByteUShift {
    static public void main(String args[]) {
        char hex[] = {
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
        };
        byte b = (byte) 0xf1;
        byte c = (byte) (b >> 4);
        byte d = (byte) (b >> 4);
        byte e = (byte) ((b & 0xff) >> 4);
        System.out.println("              b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
        System.out.println("         b >> 4 = 0x" + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
        System.out.println("         b >>> 4 = 0x" + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
    }
}
```
The following output of this program shows how the `>>>` operator appears to do nothing when dealing with bytes. The variable `b` is set to an arbitrary negative byte value for this demonstration. Then `c` is assigned the byte value of `b` shifted right by four, which is `0xff` because of the expected sign extension. Then `d` is assigned the byte value of `b` unsigned shifted right by four, which you might have expected to be `0x0f`, but is actually `0xff` because of the sign extension that happened when `b` was promoted to `int` before the shift. The last expression sets `e` to the byte value of `b` masked to 8 bits using the AND operator, then shifted right by four, which produces the expected value of `0x0f`. Notice that the unsigned shift right operator was not used for `d`, since the state of the sign bit after the AND was known.

```
 b = 0xf1
 b >> 4 = 0xff
 b >>> 4 = 0xff
 (b & 0xff) >> 4 = 0x0f
```

Bitwise Operator Compound Assignments

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation. For example, the following two statements, which shift the value in `a` right by four bits, are equivalent:

```
a = a >> 4;
```
Likewise, the following two statements, which result in a being assigned the bitwise expression a OR b, are equivalent:

```java
a = a | b;
a |= b;
```

The following program creates a few integer variables and then uses compound bitwise operator assignments to manipulate the variables:

```java
class OpBitEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;

        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

The output of this program is shown here:

```
a = 3
b = 1
c = 6
```

Relational Operators

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:
The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, `==`, and the inequality test, `!=`. Notice that in Java equality is denoted with two equal signs, not one. (Remember: a single equal sign is the assignment operator.) Only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other.

As stated, the result produced by a relational operator is a boolean value. For example, the following code fragment is perfectly valid:

```java
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of `a < b` (which is false) is stored in `c`. 

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>Equal to</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equal to</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>
If you are coming from a C/C++ background, please note the following. In C/C++, these types of statements are very common:

```java
int done;
//...
if(!done)... // Valid in C/C++
if(done)... // but not in Java.
```

In Java, these statements must be written like this:

```java
if(done == 0)... // This is Java-style.
if(done != 0)... 
```

The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero. In Java, true and false are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

**Boolean Logical Operators**
The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant boolean value.
The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state: !true == false and !false == true. The following table shows the effect of each logical operation:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Logical XOR (exclusive OR)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Short-circuit AND</td>
</tr>
<tr>
<td>!</td>
<td>Logical unary NOT</td>
</tr>
<tr>
<td>&amp;=</td>
<td>AND assignment</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>^=</td>
<td>XOR assignment</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>?::</td>
<td>Ternary if-then-else</td>
</tr>
</tbody>
</table>

Here is a program that is almost the same as the BitLogic example shown earlier, but it operates on boolean logical values instead of binary bits:

```java
// Demonstrate the boolean logical operators.
class BoolLogic {
    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
```
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("a|b = " + c);
System.out.println("a&b = " + d);
System.out.println("a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println("!a = " + g);
}
}

After running this program, you will see that the same logical rules apply to boolean values as they did to bits. As you can see from the following output, the string representation of a Java boolean value is one of the literal values true or false:

a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false

Short-Circuit Logical Operators
Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are commonly known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these
operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```java
if (denom != 0 && num / denom > 10)
```

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when denom is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```java
if(c==1 & e++ < 100) d = 100;
```

Here, using a single & ensures that the increment operation will be applied to e whether c is equal to 1 or not.

**NOTE**

The formal specification for Java refers to the short-circuit operators as the conditional-and and the
The Assignment Operator
You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

```
var = expression;
```

Here, the type of var must be compatible with the type of expression.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;

x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x. Using a “chain of assignment” is an easy way to set a group of variables to a common value.

The ? Operator
Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This
operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

    expression1 ? expression2 : expression3

Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same (or compatible) type, which can’t be void.

Here is an example of the way that the ? is employed:

    ratio = denom == 0 ? 0 : num / denom;

When Java evaluates this assignment expression, it first looks at the expression to the left of the question mark. If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression. If denom does not equal zero, then the expression after the colon is evaluated and used for the value of the entire ? expression. The result produced by the ? operator is then assigned to ratio.

Here is a program that demonstrates the ? operator. It uses it to obtain the absolute value of a variable.

    // Demonstrate ?.
    class Ternary {
        public static void main(String args[]) {
            int i, k;

            i = 10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);

i = -10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
}
}

The output generated by the program is shown here:

Absolute value of 10 is 10
Absolute value of -10 is 10

Operator Precedence

Table 4-1 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the [ ], ( ), and . can also act like operators. In that capacity, they would have the highest precedence.
Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

\[ a \gg b + 3 \]

This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this:

\[ a \gg (b + 3) \]

However, if you want to first shift a right by b positions and then add 3 to that result, you will need to parenthesize the expression like this:

\[ (a \gg b) + 3 \]
In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For anyone reading your code, a complicated expression can be difficult to understand. Adding redundant but clarifying parentheses to complex expressions can help prevent confusion later. For example, which of the following expressions is easier to read?

\[
\begin{align*}
\text{a | 4 + c} & \gg \text{b & 7} \\
(\text{a | ((4 + c) \gg b) & 7})
\end{align*}
\]

One other point: parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.
A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java’s program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion. All of Java’s control statements are examined here.

Java’s Selection Statements

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program’s execution based upon conditions known only during runtime. You will be pleasantly surprised by the power and flexibility contained in these two statements.

The if statement was introduced in Chapter 2. It is examined in detail here. The if statement is Java’s conditional branch statement. It can be used to route program execution through two different paths. Here is
the general form of the if statement:

```java
if (condition) statement1;
else statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```java
int a, b;
//...
if(a < b) a = 0;
else b = 0;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.

Most often, the expression used to control the if will involve the relational operators. However, this is not technically necessary. It is possible to control the if using a single boolean variable, as shown in this code fragment:

```java
boolean dataAvailable;
//...
if (dataAvailable)
    ProcessData();
else
    waitForMoreData();
```
Remember, only one statement can appear directly after the if or the else. If you want to include more statements, you’ll need to create a block, as in this fragment:

```java
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
} else
    waitForMoreData();
```

Here, both statements within the if block will execute if `bytesAvailable` is greater than zero.

Some programmers find it convenient to include the curly braces when using the if, even when there is only one statement in each clause. This makes it easy to add another statement at a later date, and you don’t have to worry about forgetting the braces. In fact, forgetting to define a block when one is needed is a common cause of errors. For example, consider the following code fragment:

```java
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
} else
    waitForMoreData();
    bytesAvailable = n;
```

It seems clear that the statement `bytesAvailable = n;` was intended to be executed inside the else clause, because of the indentation level. However, as you recall,
whitespace is insignificant to Java, and there is no way for the compiler to know what was intended. This code will compile without complaint, but it will behave incorrectly when run. The preceding example is fixed in the code that follows:

```java
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
} else {
    waitForMoreData();
    bytesAvailable = n;
}
```

### Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```java
if(i == 10) {
    if(j < 20) a = b;
    if(k > 100) c = d; // this if is
    else a = c;        // associated with this else
} else a = d;          // this else refers to if(i == 10)
```

As the comments indicate, the final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else
refers to if(k > 100) because it is the closest if within the same block.

The if-else-if Ladder
A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

```java
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
...
else
    statement;
```

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.
Here is a program that uses an if-else-if ladder to determine which season a particular month is in.

```java
// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String args[]) {
        int month = 4; // April
        String season;
        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
    }
}
```

Here is the output produced by the program:

April is in the Spring.

You might want to experiment with this program before moving on. As you will find, no matter what value you give month, one and only one assignment statement within the ladder will be executed.

**switch**

The switch statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative
than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
        break;
    ...
    ...
    case valueN:
        // statement sequence
        break;
    default:
        // default statement sequence
}
```

For versions of Java prior to JDK 7, expression must be of type byte, short, int, char, or an enumeration. (Enumerations are described in Chapter 12.) Beginning with JDK 7, expression can also be of type String. Each value specified in the case statements must be a unique constant expression (such as a literal value). Duplicate case values are not allowed. The type of each value must be compatible with the type of expression.

The switch statement works like this: The value of the expression is compared with each of the values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the
constants matches the value of the expression, then the default statement is executed. However, the default statement is optional. If no case matches and no default is present, then no further action is taken.

The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. This has the effect of “jumping out” of the switch.

Here is a simple example that uses a switch statement:

```java
// A simple example of the switch.
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++) {
            switch(i) {
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                    break;
                case 2:
                    System.out.println("i is two.");
                    break;
                case 3:
                    System.out.println("i is three.");
                    break;
                default:
                    System.out.println("i is greater than 3.");
            }
        }
    }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
```
i is three.
i is greater than 3.
i is greater than 3.

As you can see, each time through the loop, the statements associated with the case constant that matches i are executed. All others are bypassed. After i is greater than 3, no case statements match, so the default statement is executed.

The break statement is optional. If you omit the break, execution will continue on into the next case. It is sometimes desirable to have multiple cases without break statements between them. For example, consider the following program:

```java
// In a switch, break statements are optional.
class MissingBreak {
    public static void main(String args[]) {
        for(int i=0; i<12; i++)
            switch(i) {
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                    System.out.println("i is less than 5");
                    break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                    System.out.println("i is less than 10");
                    break;
                default:
                    System.out.println("i is 10 or more");
            }
    }
}
```
This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

As you can see, execution falls through each case until a break statement (or the end of the switch) is reached.

While the preceding example is, of course, contrived for the sake of illustration, omitting the break statement has many practical applications in real programs. To sample its more realistic usage, consider the following rewrite of the season example shown earlier. This version uses a switch to provide a more efficient implementation.
As mentioned, beginning with JDK 7, you can use a string to control a switch statement. For example,

```java
// Use a string to control a switch statement.

class StringSwitch {
    public static void main(String args[]) {
        String str = "two";

        String season;

        switch (str) {
            case "one":
            case "Two":
                season = "Winter";
                break;
            case "three":
            case "Three":
                season = "Spring";
                break;
            case "four":
            case "Four":
                season = "Summer";
                break;
            case "five":
            case "Five":
                season = "Autumn";
                break;
            default:
                season = "Bogus Month";
        }
        System.out.println("April is in the " + season + ".");
    }
}
```

As mentioned, beginning with JDK 7, you can use a string to control a switch statement. For example,
As you would expect, the output from the program is

two

The string contained in str (which is “two” in this program) is tested against the case constants. When a match is found (as it is in the second case), the code sequence associated with that sequence is executed.

Being able to use strings in a switch statement streamlines many situations. For example, using a string-based switch is an improvement over using the equivalent sequence of if/else statements. However, switching on strings is more expensive than switching on integers. Therefore, it is best to switch on strings only in cases in which the controlling data is already in string form. In other words, don’t use strings in a switch unnecessarily.

Nested switch Statements
You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch. For example, the following fragment is perfectly valid:

```java
switch(count) {
    case 1:
        switch(target) { // nested switch
            case 0:
                System.out.println("target is zero");
                break;
            case 1: // no conflicts with outer switch
                System.out.println("target is one");
                break;
        }
        break;
    case 2: // ...
}
```

Here, the case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch. The count variable is compared only with the list of cases at the outer level. If count is 1, then target is compared with the inner list cases.

In summary, there are three important features of the switch statement to note:

- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- No two case constants in the same switch can have
identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.

- A switch statement is usually more efficient than a set of nested if statements.

The last point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a switch statement, the Java compiler will inspect each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses. The compiler can do this because it knows that the case constants are all the same type and simply must be compared for equality with the switch expression. The compiler has no such knowledge of a long list of if expressions.

Iteration Statements
Java’s iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.
The while loop is Java’s most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```java
while(condition) {
    // body of loop
}
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Here is a while loop that counts down from 10, printing exactly ten lines of “tick”:

```java
// Demonstrate the while loop.
class While {
    public static void main(String args[]) {
        int n = 10;

        while(n > 0) {
            System.out.println("tick " + n);
            n--;
        }
    }
}
```

When you run this program, it will “tick” ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
```
Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to println() is never executed:

```java
int a = 10, b = 20;
while(a > b)
    System.out.println("This will not be displayed");
```

The body of the while (or any other of Java’s loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

```java
// The target of a loop can be empty.
class NoBody {
    public static void main(String args[]) {
        int i, j;

        i = 100;
        j = 200;

        // find midpoint between i and j
        while(++i < --j); // no body in this loop

        System.out.println("Midpoint is " + i);
    }
}
```

This program finds the midpoint between i and j. It generates the following output:

```
Midpoint is 150
```
Here is how this while loop works. The value of i is incremented, and the value of j is decremented. These values are then compared with one another. If the new value of i is still less than the new value of j, then the loop repeats. If i is equal to or greater than j, the loop stops. Upon exit from the loop, i will hold a value that is midway between the original values of i and j. (Of course, this procedure only works when i is less than j to begin with.) As you can see, there is no need for a loop body; all of the action occurs within the conditional expression, itself. In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while

As you just saw, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is
do {
    // body of loop
} while (condition);

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java’s loops, condition must be a Boolean expression.

Here is a reworked version of the “tick” program that demonstrates the do-while loop. It generates the same output as before.

```java
// Demonstrate the do-while loop.
class DoWhile {
    public static void main(String args[]) {
        int n = 10;

        do {
            System.out.println("tick " + n);
            n--;
        } while(n > 0);
    }
}
```

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

```java
do {
    System.out.println("tick " + n);
} while(--n > 0);
```

In this example, the expression `(--n > 0)` combines the decrement of `n` and the test for zero into one expression. Here is how it works. First, the `--n` statement executes, decrementing `n` and returning the new value of `n`. This
value is then compared with zero. If it is greater than zero, the loop continues; otherwise, it terminates.

The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once. Consider the following program, which implements a very simple help system for Java’s selection and iteration statements:

```java
// Using a do-while to process a menu selection
class Menu {
    public static void main(String args[])
        throws java.io.IOException {
        char choice;
        do {
            System.out.println("Help on: ");
            System.out.println("  1. if");
            System.out.println("  2. switch");
            System.out.println("  3. while");
            System.out.println("  4. do-while");
            System.out.println("  5. for\n");
            System.out.println("Choose one:");

            choice = (char) System.in.read();
        } while( choice < '1' || choice > '5');

        System.out.println("\n");

        switch(choice) {
        case '1':
            System.out.println("The if:\n");
            System.out.println("if(condition) statement;";
            System.out.println("else statement;");
            break;
        case '2':
```
Here is a sample run produced by this program:

Help on:
1. if
2. switch
3. while
4. do-while
5. for

Choose one:
4
The do-while:
do {
   statement;
} while (condition);

In the program, the do-while loop is used to verify that
the user has entered a valid choice. If not, then the user is reprompted. Since the menu must be displayed at least once, the do-while is the perfect loop to accomplish this.

A few other points about this example: Notice that characters are read from the keyboard by calling System.in.read(). This is one of Java’s console input functions. Although Java’s console I/O methods won’t be discussed in detail until Chapter 13, System.in.read() is used here to obtain the user’s choice. It reads characters from standard input (returned as integers, which is why the return value was cast to char). By default, standard input is line buffered, so you must press ENTER before any characters that you type will be sent to your program.

Java’s console input can be a bit awkward to work with. Further, most real-world Java programs will be graphical and window-based. For these reasons, not much use of console input has been made in this book. However, it is useful in this context. One other point to consider: Because System.in.read() is being used, the program must specify the throws java.io.IOException clause. This line is necessary to handle input errors. It is part of Java’s exception handling features, which are discussed in Chapter 10.

You were introduced to a simple form of the for loop in Chapter 2. As you will see, it is a powerful and versatile
Beginning with JDK 5, there are two forms of the for loop. The first is the traditional form that has been in use since the original version of Java. The second is the new “for-each” form. Both types of for loops are discussed here, beginning with the traditional form.

Here is the general form of the traditional for statement:

```java
for(initialization; condition; iteration) {
  // body
}
```

If only one statement is being repeated, there is no need for the curly braces.

The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is executed only once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then...
executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Here is a version of the “tick” program that uses a for loop:

```java
// Demonstrate the for loop.
class ForTick {
    public static void main(String args[]) {
        int n;

        for(n=10; n>0; n--)
            System.out.println("tick "+n);
    }
}
```

**Declaring Loop Control Variables Inside the for Loop**

Often the variable that controls a for loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the for. For example, here is the preceding program recoded so that the loop control variable `n` is declared as an int inside the for:

```java
// Declare a loop control variable inside the for.
class ForTick {
    public static void main(String args[]) {
        // here, n is declared inside of the for loop
        for(int n=10; n>0; n--)
            System.out.println("tick "+n);
    }
}
```
When you declare a variable inside a for loop, there is one important point to remember: the scope of that variable ends when the for statement does. (That is, the scope of the variable is limited to the for loop.) Outside the for loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the for loop.

When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the for. For example, here is a simple program that tests for prime numbers. Notice that the loop control variable, i, is declared inside the for since it is not needed elsewhere.

```java
// Test for primes.
class FindPrime {
    public static void main(String args[]) {
        int num;
        boolean isPrime;

        num = 14;

        if(num < 2) isPrime = false;
        else isPrime = true;

        for(int i=2; i <= num/i; i++) {
            if((num % i) == 0) {
                isPrime = false;
                break;
            }
        }

        if(isPrime) System.out.println("Prime");
        else System.out.println("Not Prime");
    }
}
```
Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the for loop. For example, consider the loop in the following program:

```java
class Sample {
    public static void main(String args[]) {
        int a, b;

        b = 4;
        for(a=1; a<b; a++) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            b--;
        }
    }
}
```

As you can see, the loop is controlled by the interaction of two variables. Since the loop is governed by two variables, it would be useful if both could be included in the for statement, itself, instead of b being handled manually. Fortunately, Java provides a way to accomplish this. To allow two or more variables to control a for loop, Java permits you to include multiple statements in both the initialization and iteration portions of the for. Each statement is separated from the next by a comma.

Using the comma, the preceding for loop can be more efficiently coded, as shown here:

```java
// Using the comma.
class Comma {
    public static void main(String args[]) {
```

```java
  int a, b;
  b = 4;
  for(a=1; a<b; a++) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      b--;
  }
}
```
In this example, the initialization portion sets the values of both \( a \) and \( b \). The two comma-separated statements in the iteration portion are executed each time the loop repeats. The program generates the following output:

\[
\begin{align*}
  a &= 1 \\
  b &= 4 \\
  a &= 2 \\
  b &= 3
\end{align*}
\]

NOTE

If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java. In Java, the comma is a separator.

Some for Loop Variations
The for loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts—the initialization, the conditional test, and the iteration—do not need to be used for only those purposes. In fact, the three sections of the for can be used for any purpose you desire. Let's look at some examples.
One of the most common variations involves the conditional expression. Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the for can be any Boolean expression. For example, consider the following fragment:

```java
boolean done = false;
for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

In this example, the for loop continues to run until the boolean variable done is set to true. It does not test the value of i.

Here is another interesting for loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

```java
// Parts of the for loop can be empty.
class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;

        i = 0;
        for( ; !done; ) {
            System.out.println("i is " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

Here, the initialization and iteration expressions have been moved out of the for. Thus, parts of the for are
empty. While this is of no value in this simple example—indeed, it would be considered quite poor style—there can be times when this type of approach makes sense. For example, if the initial condition is set through a complex expression elsewhere in the program or if the loop control variable changes in a nonsequential manner determined by actions that occur within the body of the loop, it may be appropriate to leave these parts of the for empty.

Here is one more for loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the for empty. For example:

```
for( ; ; ) {
    // …
}
```

This loop will run forever because there is no condition under which it will terminate. Although there are some programs, such as operating system command processors, that require an infinite loop, most “infinite loops” are really just loops with special termination requirements. As you will soon see, there is a way to terminate a loop—even an infinite loop like the one shown—that does not make use of the normal loop conditional expression.

The For-Each Version of the for Loop

Beginning with JDK 5, a second form of for was defined that implements a “for-each” style loop. As you may
know, contemporary language theory has embraced the for-each concept, and it is quickly becoming a standard feature that programmers have come to expect. A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword foreach, Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of for is also referred to as the enhanced for loop.

The general form of the for-each version of the for is shown here:

```
for(type itr-var: collection) statement-block
```

Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by collection. There are various types of collections that can be used with the for, but the only type used in this chapter is the array. (Other types of collections that can be used with the for, such as those defined by the Collections Framework, are discussed later in this book.) With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the
Because the iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, type must be compatible with the element type of the array.

To understand the motivation behind a for-each style loop, consider the type of for loop that it is designed to replace. The following fragment uses a traditional for loop to compute the sum of the values in an array:

```c
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

To compute the sum, each element in `nums` is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the `nums` array by `i`, the loop control variable.

The for-each style for automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the for:

```c
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```
for(int x: nums) sum += x;

With each pass through the loop, x is automatically given a value equal to the next element in nums. Thus, on the first iteration, x contains 1; on the second iteration, x contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

Here is an entire program that demonstrates the for-each version of the for just described:

```
// Use a for-each style for loop.
class ForEach {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        // use for-each style for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: "+ x);
            sum += x;
        }

        System.out.println("Summation: "+ sum);
    }
}
```

The output from the program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```
As this output shows, the for-each style for automatically cycles through an array in sequence from the lowest index to the highest.

Although the for-each for loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a break statement. For example, this program sums only the first five elements of `nums`:

```java
// Use break with a for-each style for.
class ForEach2 {
    public static void main(String args[]) {
        int sum = 0;

        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // use for to display and sum the values
        for(int x : nums) {
            System.out.println("Value is: "+x);
            sum += x;
            if(x == 5) break; // stop the loop when 5 is obtained
        }
        System.out.println("Summation of first 5 elements: "+sum);
    }
}
```

This is the output produced:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15
```

As is evident, the for loop stops after the fifth element has been obtained. The break statement can also be used with Java’s other loops, and it is discussed in detail later.
There is one important point to understand about the for-each style loop. Its iteration variable is “read-only” as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can’t change the contents of the array by assigning the iteration variable a new value. For example, consider this program:

```java
// The for-each loop is essentially read-only.
class NoChange {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        for(int x: nums) {
            System.out.print(x + " ");
            x = x * 10; // no effect on nums
        }

        System.out.println();

        for(int x : nums)
            System.out.print(x + " ");

        System.out.println();
    }
}
```

The first for loop increases the value of the iteration variable by a factor of 10. However, this assignment has no effect on the underlying array nums, as the second for loop illustrates. The output, shown here, proves this point:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Iterating Over Multidimensional Arrays
The enhanced version of the for also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of arrays of arrays. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array, because each iteration obtains the next array, not an individual element. Furthermore, the iteration variable in the for loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each for to iterate over an array of N dimensions, the objects obtained will be arrays of N–1 dimensions. To understand the implications of this, consider the following program. It uses nested for loops to obtain the elements of a two-dimensional array in row-order, from first to last.
The output from this program is shown here:

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90

In the program, pay special attention to this line:

for(int x[]: nums) {

Notice how x is declared. It is a reference to a one-
dimensional array of integers. This is necessary because each iteration of the for obtains the next array in nums, beginning with the array specified by nums[0]. The inner for loop then cycles through each of these arrays, displaying the values of each element.

Applying the Enhanced for
Since the for-each style for can only cycle through an array sequentially, from start to finish, you might think that its use is limited, but this is not true. A large number of algorithms require exactly this mechanism. One of the most common is searching. For example, the following program uses a for loop to search an unsorted array for a value. It stops if the value is found.

```java
// Search an array using for-each style for.
class Search {
    public static void main(String args[]) {
        int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
        int val = 5;
        boolean found = false;

        // use for-each style for to search nums for val
        for(int x : nums) {
            if(x == val) {
                found = true;
                break;
            }
        }

        if(found)
            System.out.println("Value found!");
    }
}
```

The for-each style for is an excellent choice in this application because searching an unsorted array involves
examining each element in sequence. (Of course, if the array were sorted, a binary search could be used, which would require a different style loop.) Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

Although we have been using arrays in the examples in this chapter, the for-each style for is especially useful when operating on collections defined by the Collections Framework, which is described in Part II. More generally, the for can cycle through the elements of any collection of objects, as long as that collection satisfies a certain set of constraints, which are described in Chapter 17.

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests for loops:

```java
// Loops may be nested.
class Nested {
    public static void main(String args[]) {
        int i, j;

        for(i=0; i<10; i++) {
            for(j=i; j<10; j++)
                System.out.print(".");
            System.out.println();
        }
    }
}
```

The output produced by this program is shown here:
Jump Statements

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of your program. Each is examined here.

NOTE

In addition to the jump statements discussed here, Java supports one other way that you can change your program’s flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords try, catch, throw, throws, and finally. In essence, the exception handling mechanism allows your program to perform a nonlocal branch. Since exception handling is a large topic, it is discussed in its own chapter, Chapter 10.

Using break

In Java, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it
can be used as a “civilized” form of goto. The last two uses are explained here.

Using break to Exit a Loop
By using `break`, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a `break` statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```java
// Using break to exit a loop.
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

As you can see, although the `for` loop is designed to run from 0 to 99, the `break` statement causes it to terminate early, when `i` equals 10.
The break statement can be used with any of Java’s loops, including intentionally infinite loops. For example, here is the preceding program coded by use of a while loop. The output from this program is the same as just shown.

```java
// Using break to exit a while loop.
class BreakLoop2 {
    public static void main(String args[]) {
        int i = 0;

        while(i < 100) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
            i++;
        }
        System.out.println("Loop complete.");
    }
}
```

When used inside a set of nested loops, the break statement will only break out of the innermost loop. For example:

```java
// Using break with nested loops.
class BreakLoop3 {
    public static void main(String args[]) {
        for(int i=0; i< 3; i++) {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break; // terminate loop if j is 10
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Loops complete.");
    }
}
```

This program generates the following output:
As you can see, the break statement in the inner loop only causes termination of that loop. The outer loop is unaffected.

Here are two other points to remember about break. First, more than one break statement may appear in a loop. However, be careful. Too many break statements have the tendency to destruct your code. Second, the break that terminates a switch statement affects only that switch statement and not any enclosing loops.

REMEMBER

break was not designed to provide the normal means by which a loop is terminated. The loop’s conditional expression serves this purpose. The break statement should be used to cancel a loop only when some sort of special situation occurs.

Using break as a Form of Goto

In addition to its uses with the switch statement and loops, the break statement can also be employed by itself to provide a “civilized” form of the goto statement. Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations. There are, however, a few places where
the goto is a valuable and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the break statement. By using this form of break, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a switch. They can be any block. Further, you can specify precisely where execution will resume, because this form of break works with a label. As you will see, break gives you the benefits of a goto without its problems.

The general form of the labeled break statement is shown here:

```
break label;
```

Most often, label is the name of a label that identifies a block of code. This can be a standalone block of code but it can also be a block that is the target of another statement. When this form of break executes, control is transferred out of the named block. The labeled block must enclose the break statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled break statement to exit from a set of nested blocks. But you cannot use break to transfer control out of a block that does not enclose the break statement.

To name a block, put a label at the start of it. A label is any valid Java identifier followed by a colon. Once
you have labeled a block, you can then use this label as the target of a break statement. Doing so causes execution to resume at the end of the labeled block. For example, the following program shows three nested blocks, each with its own label. The break statement causes execution to jump forward, past the end of the block labeled second, skipping the two println( ) statements.

// Using break as a civilized form of goto.
class Break {
    public static void main(String args[]) {
        boolean t = true;

        first: {
            second: {
                third: {
                    System.out.println("Before the break.");
                    if(t) break second; // break out of second block
                    System.out.println("This won't execute");
                }
                System.out.println("This won't execute");
            }
            System.out.println("This is after second block.");
        }
    }
}

Running this program generates the following output:

Before the break.
This is after second block.

One of the most common uses for a labeled break statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:

// Using break to exit from nested loops
class BreakLoop4 {
    public static void main(String args[]) {
        outer: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
            for(int j=0; j< 100; j++) {
                if(j == 10) break outer; // exit both loops
                System.out.print(j + " ");
            }
            System.out.println("This will not print");
        }
        System.out.println("Loops complete.");
    }
}

This program generates the following output:

    Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

As you can see, when the inner loop breaks to the outer loop, both loops have been terminated. Notice that this example labels the for statement, which has a block of code as its target.

    Keep in mind that you cannot break to any label which is not defined for an enclosing block. For example, the following program is invalid and will not compile:

    // This program contains an error.
    class BreakErr {
        public static void main(String args[]) {
            one: for(int i=0; i<3; i++) {
                System.out.print("Pass " + i + ": ");
            }
            for(int j=0; j<100; j++) {
                if(j == 10) break one; // WRONG
                System.out.print(j + " ");
            }
        }
    }
Since the loop labeled one does not enclose the break statement, it is not possible to transfer control out of that block.

Using continue
Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses continue to cause two numbers to be printed on each line:

```java
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println(" ");
    }
  }
}
```

This code uses the % operator to check if i is even. If it
is, the loop continues without printing a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

As with the `break` statement, `continue` may specify a label to describe which enclosing loop to continue. Here is an example program that uses `continue` to print a triangular multiplication table for 0 through 9:

```
// Using continue with a label.
class ContinueLabel {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" "+(i * j));
            }
            System.out.println();
        }
    }
}
```

The `continue` statement in this example terminates the loop counting `j` and continues with the next iteration of the loop counting `i`. Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
```
Good uses of continue are rare. One reason is that Java provides a rich set of loop statements which fit most applications. However, for those special circumstances in which early iteration is needed, the continue statement provides a structured way to accomplish it.

return
The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of return must wait until methods are discussed in Chapter 6, a brief look at return is presented here.

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, return causes execution to return to the Java run-time system, since it is the run-time system that calls main():

```java
// Demonstrate return.
class Return {
    public static void main(String args[]) {
        boolean t = true;
        System.out.println("Before the return.");
```
if(t) return; // return to caller

System.out.println("This won't execute.");
}
}

The output from this program is shown here:

Before the return.

As you can see, the final println( ) statement is not executed. As soon as return is executed, control passes back to the caller.

One last point: In the preceding program, the if(t) statement is necessary. Without it, the Java compiler would flag an “unreachable code” error because the compiler would know that the last println( ) statement would never be executed. To prevent this error, the if statement is used here to trick the compiler for the sake of this demonstration.
CHAPTER 6
Introducing Classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the this keyword.

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the main( ) method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is
an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class’ code defines the interface to its data.

A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:
The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class’ data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn.
All methods have the same general form as main( ), which we have been using thus far. However, most methods will not be specified as static or public. Notice that the general form of a class does not specify a main( ) method. Java classes do not need to have a main( ) method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications, such as applets, don’t require a main( ) method at all.

NOTE
C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately. This sometimes makes for very large .java files, since any class must be entirely defined in a single source file. This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain.

A Simple Class
Let’s begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods (but some will be added soon).

```java
class Box {
    double width;
```
As stated, a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type Box to come into existence.

To actually create a Box object, you will use a statement like the following:

```java
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, mybox will be an instance of Box. Thus, it will have “physical” reality. For the moment, don’t worry about the details of this statement.

As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every Box object will contain its own copies of the instance variables width, height, and depth. To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the width variable of mybox the value 100, you would use the following statement:

```java
mybox.width = 100;
```
This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object. One other point: Although commonly referred to as the dot operator, the formal specification for Java categorizes the . as a separator. However, since the use of the term “dot operator” is widespread, it is used in this book.

Here is a complete program that uses the Box class:
You should call the file that contains this program BoxDemo.java, because the main( ) method is in the class called BoxDemo, not the class called Box. When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo. The Java compiler automatically puts each class into its own .class file. It is not necessary for both the Box and the BoxDemo class to actually be in the same source file. You could put each class in its own file, called Box.java and BoxDemo.java, respectively.
To run this program, you must execute BoxDemo.class. When you do, you will see the following output:

Volume is 3000.0

As stated earlier, each object has its own copies of the instance variables. This means that if you have two Box objects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two Box objects:
The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, mybox1’s data is completely separate
from the data contained in mybox2.

Declaring Objects
As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let’s look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

```java
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```java
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares mybox as a reference to an object
of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object. Any attempt to use mybox at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds the memory address of the actual Box object. The effect of these two lines of code is depicted in Figure 6-1.

NOTE

Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers. This suspicion is, essentially, correct. An object reference is similar to a memory pointer. The main difference—and the key to Java’s safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

A Closer Look at new

As just explained, the new operator dynamically allocates memory for an object. It has this general form:

    class-var = new classname ( );
Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with Box. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

At this point, you might be wondering why you do not need to use new for such things as integers or characters.
The answer is that Java’s primitive types are not implemented as objects. Rather, they are implemented as “normal” variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently. Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.

It is important to understand that new allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that new will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle exceptions in Chapter 10.) For the sample programs in this book, you won’t need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Let’s once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical
framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

Assigning Object Reference Variables
Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```java
Box b1 = new Box();
Box b2 = b1;
```

You might think that `b2` is being assigned a reference to a copy of the object referred to by `b1`. That is, you might think that `b1` and `b2` refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, `b1` and `b2` will both refer to the same object. The assignment of `b1` to `b2` did not allocate any memory or copy any part of the original object. It simply makes `b2` refer to the same object as does `b1`. Thus, any changes made to the object through `b2` will affect the object to which `b1` is referring, since they are the same object.

This situation is depicted here:
Although \( b_1 \) and \( b_2 \) both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to \( b_1 \) will simply unhook \( b_1 \) from the original object without affecting the object or affecting \( b_2 \). For example:

```java
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, \( b_1 \) has been set to null, but \( b_2 \) still points to the original object.

**REMEMBER**

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

**Introducing Methods**

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact,
much of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

```java
    type name(parameter-list) {
        // body of method
    }
```

Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void. The name of the method is specified by name. This can be any legal identifier other than those already used by other items within the current scope. The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

```java
    return value;
```

Here, value is the value returned.
In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let’s begin by adding a method to the Box class. It may have occurred to you while looking at the preceding programs that the computation of a box’s volume was something that was best handled by the Box class rather than the BoxDemo class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the Box class compute it. To do this, you must add a method to Box, as shown here:
This program includes a method inside the box class.

class Box {
    double width;
    double height;
    double depth;

    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // display volume of first box
        mybox1.volume();

        // display volume of second box
        mybox2.volume();
    }
}

This program generates the following output, which is
the same as the previous version.

    Volume is 3000.0
    Volume is 162.0
Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the `volume()` method on `mybox1`. That is, it calls `volume()` relative to the `mybox1` object, using the object’s name followed by the dot operator. Thus, the call to `mybox1.volume()` displays the volume of the box defined by `mybox1`, and the call to `mybox2.volume()` displays the volume of the box defined by `mybox2`. Each time `volume()` is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When `mybox1.volume()` is executed, the Java runtime system transfers control to the code defined inside `volume()`. After the statements inside `volume()` have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java’s way of implementing subroutines.

There is something very important to notice inside the `volume()` method: the instance variables `width`, `height`, and `depth` are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always
invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that width, height, and depth inside volume( ) implicitly refer to the copies of those variables found in the object that invokes volume( ).

Let’s review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

Returning a Value

While the implementation of volume( ) does move the computation of a box’s volume inside the Box class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement volume( ) is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:
As you can see, when volume( ) is called, it is put on the right side of an assignment statement. On the left is a variable, in this case vol, that will receive the value returned by volume( ). Thus, after
There are two important things to understand about returning values:

• The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is boolean, you could not return an integer.
• The variable receiving the value returned by a method (such as `vol`, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the `vol` variable. The call to `volume()` could have been used in the `println()` statement directly, as shown here:

```java
System.out.println("Volume is" + mybox1.volume());
```

In this case, when `println()` is executed, `mybox1.volume()` will be called automatically and its value will be passed to `println()`.

Adding a Method That Takes Parameters

While some methods don’t need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data
and/or be used in a number of slightly different situations. To illustrate this point, let’s use a very simple example. Here is a method that returns the square of the number 10:

```java
int square()
{
    return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make `square()` much more useful.

```java
int square(int i)
{
    return i * i;
}
```

Now, `square()` will return the square of whatever value it is called with. That is, `square()` is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```java
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

In the first call to `square()`, the value 5 will be passed into parameter `i`. In the second call, `i` will receive the value 9. The third invocation passes the value of `y`,
which is 2 in this example. As these examples show, \( \text{square}( ) \) is able to return the square of whatever data it is passed.

It is important to keep the two terms parameter and argument straight. A parameter is a variable defined by a method that receives a value when the method is called. For example, in \( \text{square}( ) \), \( i \) is a parameter. An argument is a value that is passed to a method when it is invoked. For example, \( \text{square}(100) \) passes 100 as an argument. Inside \( \text{square}( ) \), the parameter \( i \) receives that value.

You can use a parameterized method to improve the Box class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```java
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can’t change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable
appropriately. This concept is implemented by the following program:

```java
// This program uses a parameterized method.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }

    // sets dimensions of box
    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}

class BoxDemo5 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

As you can see, the setDim( ) method is used to set the
dimensions of each box. For example, when

mybox1.setDim(10, 20, 15);

is executed, 10 is copied into parameter w, 20 is copied into h, and 15 is copied into d. Inside setDim( ) the values of w, h, and d are then assigned to width, height, and depth, respectively.

For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

Constructors
It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like setDim( ), it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called
immediately after the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even \texttt{void}. This is because the implicit return type of a class’ constructor is the class type itself. It is the constructor’s job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the \texttt{Box} example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace \texttt{setDim( )} with a constructor.

Let’s begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:
When this program is run, it generates the following results:

Constructing Box
Constructing Box
Volume is 1000.0
As you can see, both mybox1 and mybox2 were initialized by the Box( ) constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both mybox1 and mybox2 will have the same volume. The println( ) statement inside Box( ) is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let’s reexamine the new operator. As you know, when you allocate an object, you use the following general form:

    class-var = new classname ( );

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

    Box mybox1 = new Box();

new Box( ) is calling the Box( ) constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of Box that did not define a constructor. The default constructor automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won’t do for
more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the Box( ) constructor in the preceding example does initialize a Box object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct Box objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes it much more useful. For example, the following version of Box defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created.
The output from this program is shown here:

Volume is 3000.0
Volume is 162.0

As you can see, each object is initialized as specified
in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the `Box( )` constructor when `new` creates the object. Thus, `mybox1`'s copy of width, height, and depth will contain the values 10, 20, and 15, respectively.

The `this` Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the `this` keyword. `this` can be used inside any method to refer to the current object. That is, `this` is always a reference to the object on which the method was invoked. You can use `this` anywhere a reference to an object of the current class' type is permitted.

To better understand what `this` refers to, consider the following version of `Box( )`:

```
// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

This version of `Box( )` operates exactly like the earlier version. The use of `this` is redundant, but perfectly correct. Inside `Box( )`, `this` will always refer to the invoking object. While it is redundant in this case, `this` is useful in other contexts, one of which is explained in the
Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class’ instance variables. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class. If they had been, then width, for example, would have referred to the formal parameter, hiding the instance variable width. While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

```java
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```
A word of caution: The use of this in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use this to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

Garbage Collection
Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.
The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize( ) method on the object.

The finalize( ) method has this general form:

protected void finalize( )
{
    // finalization code here
}

Here, the keyword protected is a specifier that prevents
access to \texttt{finalize()} by code defined outside its class. This and the other access modifiers are explained in \textit{Chapter 7}.

It is important to understand that \texttt{finalize()} is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—\texttt{finalize()} will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on \texttt{finalize()} for normal program operation.

\textbf{NOTE}

If you are familiar with \texttt{C++}, then you know that \texttt{C++} allows you to define a destructor for a class, which is called when an object goes out-of-scope. \texttt{Java} does not support this idea or provide for destructors. The \texttt{finalize()} method only approximates the function of a destructor. As you get more experienced with \texttt{Java}, you will see that the need for destructor functions is minimal because of \texttt{Java}'s garbage collection subsystem.

\textbf{A Stack Class}

While the \texttt{Box} class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP)
presented in Chapter 2, one of OOP’s most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class’ data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a “data engine.” No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

To see a practical application of the preceding discussion, let’s develop one of the archetypal examples of encapsulation: the stack. A stack stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called push and pop. To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.
Here is a class called `Stack` that implements a stack for up to ten integers:

```java
// This class defines an integer stack that can hold 10 values
class Stack {
    int stck[] = new int[10];
    int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

As you can see, the `Stack` class defines two data items and three methods. The stack of integers is held by the array `stck`. This array is indexed by the variable `tos`, which always contains the index of the top of the stack. The `Stack()` constructor initializes `tos` to -1, which indicates an empty stack. The method `push()` puts an item on the stack. To retrieve an item, call `pop()`. Since
access to the stack is through \texttt{push( )} and \texttt{pop( )}, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by \texttt{push( )} and \texttt{pop( )} would remain the same.

The class \texttt{TestStack}, shown here, demonstrates the \texttt{Stack} class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```java
class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();
        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
```
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10

As you can see, the contents of each stack are separate.

One last point about the Stack class. As it is currently implemented, it is possible for the array that holds the stack, stck, to be altered by code outside of the Stack class. This leaves Stack open to misuse or mischief. In the next chapter, you will see how to remedy this situation.
CHAPTER 7
A Closer Look at Methods and Classes

This chapter continues the discussion of methods and classes begun in the preceding chapter. It examines several topics relating to methods, including overloading, parameter passing, and recursion. The chapter then returns to the class, discussing access control, the use of the keyword static, and one of Java’s most important built-in classes: String.

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java’s most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an
overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```java
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for one integer parameter.
    void test(int a) {
        System.out.println("a: " + a);
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }

    // Overload test for a double parameter
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

This program generates the following output:

No parameters
a: 10
As you can see, test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method’s parameters. However, this match need not always be exact. In some cases, Java’s automatic type conversions can play a role in overload resolution. For example, consider the following program:
// Automatic type conversions apply to overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: "+ a + " "+ b);
    }

    // Overload test for a double parameter
    void test(double a) {
        System.out.println("Inside test(double) a: "+ a);
    }
}

class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;

        ob.test();
        ob.test(10, 20);

        ob.test(i); // this will invoke test(double)
        ob.test(123.2); // this will invoke test(double)
    }
}

This program generates the following output:

No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2

As you can see, this version of OverloadDemo does not define test(int). Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found. However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call. Therefore, after test(int) is not found, Java elevates i to double and then calls test(double). Of course, if test(int) had been defined, it would have been
called instead. Java will employ its automatic type conversions only if no exact match is found.

Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function abs( ) returns the absolute value of an integer, labs( ) returns the absolute value of a long integer, and fabs( ) returns the absolute value of a floating-point value. Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java’s standard class library includes an absolute value method, called abs( ). This method is overloaded by Java’s Math class to handle all numeric types. Java determines which version of abs( ) to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name abs represents the general action that is being
performed. It is left to the compiler to choose the right specific version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name `sqr` to create methods that return the square of an integer and the square root of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the `Box` class developed in the preceding chapter. Following is the latest version of `Box`:

```java
class Box {
    double width;
    double height;
```
double depth;

// This is the constructor for Box.
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}

// compute and return volume
double volume() {
    return width * height * depth;
}

As you can see, the Box( ) constructor requires three parameters. This means that all declarations of Box objects must pass three arguments to the Box( ) constructor. For example, the following statement is currently invalid:

Box ob = new Box();

Since Box( ) requires three arguments, it’s an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the Box class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the Box constructor so that it handles the situations just described. Here is a program that contains an improved version of Box that does just that:
/* Here, Box defines three constructors to initialize the dimensions of a box various ways. */

class Box {
  double width;
  double height;
  double depth;

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
The output produced by this program is shown here:

Volume of mybox1 is 3000.0
Volume of mybox2 is −1.0
Volume of mycube is 343.0

As you can see, the proper overloaded constructor is called based upon the parameters specified when new is executed.

Using Objects as Parameters
So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```java
// Objects may be passed to methods.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }
}
```
This program generates the following output:

ob1 == ob2: true
ob1 == ob3: false

As you can see, the equals( ) method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns true. Otherwise, it returns false. Notice that the parameter o in equals( ) specifies Test as its type. Although Test is a class type created by the program, it is used in just the same way as Java’s built-in types.

One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of Box
allows one object to initialize another:

```java
// Here, Box allows one object to initialize another.

class Box {
    double width;
    double height;
    double depth;

    // Notice this constructor. It takes an object of type Box.
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```
As you will see when you begin to create your own classes, providing many forms of constructors is usually required to allow objects to be constructed in a convenient and efficient manner.

A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value

```java
class OverloadCons2 {
    public static void main(String args[]) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        Box myclone = new Box(mybox1); // create copy of mybox1

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of cube is " + vol);

        // get volume of clone
        vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);
    }
}
```
of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```java
// Primitive types are passed by value.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a and b before call: " + a + " " + b);

        ob.meth(a, b);

        System.out.println("a and b after call: " + a + " " + b);
    }
}

The output from this program is shown here:

a and b before call: 15 20
a and b after call: 15 20
```
As you can see, the operations that occur inside `meth()` have no effect on the values of `a` and `b` used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program:

```java
// Objects are passed through their references.

class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // pass an object
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
        ob.meth(ob);
    }
}
```
This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside `meth()` have affected the object used as an argument.

**REMEMBER**

When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

**Returning Objects**

A method can return any type of data, including class types that you create. For example, in the following program, the `incrByTen()` method returns an object in which the value of `a` is ten greater than it is in the invoking object.

```java
// Returning an object.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}
```
class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
                          + ob2.a);
    }
}

The output generated by this program is shown here:

ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22

As you can see, each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using new, you don’t need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Recursion
Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.
The classic example of recursion is the computation of the factorial of a number. The factorial of a number $N$ is the product of all the whole numbers between 1 and $N$. For example, 3 factorial is $1 \times 2 \times 3 \times$, or 6. Here is how a factorial can be computed by use of a recursive method:

```java
// A simple example of recursion.
class Factorial {
    // this is a recursive method
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();

        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

The output from this program is shown here:

Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

If you are unfamiliar with recursive methods, then the operation of fact( ) may seem a bit confusing. Here is how it works. When fact( ) is called with an argument of 1, the function returns 1; otherwise, it returns the product of fact(n–1)*n. To evaluate this expression, fact( ) is called with n–1. This process repeats until n equals 1 and the calls to the method begin returning.
To better understand how the `fact()` method works, let’s go through a short example. When you compute the factorial of 3, the first call to `fact()` will cause a second call to be made with an argument of 2. This invocation will cause `fact()` to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of `n` in the second invocation). This result (which is 2) is then returned to the original invocation of `fact()` and multiplied by 3 (the original value of `n`). This yields the answer, 6. You might find it interesting to insert `println()` statements into `fact()`, which will show at what level each call is and what the intermediate answers are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to “telescope” out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.
The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed. If you don’t do this, once you call the method, it will never return. This is a very common error in working with recursion. Use println( ) statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method printArray( ) prints the first i elements in the array values.

```java
// Another example that uses recursion.

class RecTest {
    int values[];
    RecTest(int i) {
        values = new int[i];
    }

    // display array -- recursively
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println("[" + (i−1) + "] " + values[i−1]);
    }
}

class Recursion2 {
    public static void main(String args[]) {
        RecTest ob = new RecTest(10);
        int i;
```
for(i=0; i<10; i++) ob.values[i] = i;

ob.printArray(10);

This program generates the following output:

[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9

Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal. For example, consider the Stack class shown at the end of Chapter 6. While it is true that the methods push() and pop() do provide a controlled interface to the stack, this interface is not enforced. That is, it is possible for another part of the program to bypass these methods and access the
stack directly. Of course, in the wrong hands, this could lead to trouble. In this section, you will be introduced to the mechanism by which you can precisely control access to the various members of a class.

How a member can be accessed is determined by the access modifier attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.) These parts of Java’s access control mechanism will be discussed later. Here, let’s begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

Java’s access modifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved. The other access modifiers are described next.

Let’s begin by defining public and private. When a member of a class is modified by public, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. Now you can understand why main( ) has always been preceded by the public modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.)

In the classes developed so far, all members of a class
have used the default access mode, which is essentially public. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

An access modifier precedes the rest of a member’s type specification. That is, it must begin a member’s declaration statement. Here is an example:

```java
public int i;
private double j;
private int myMethod(int a, char b) { //…
```

To understand the effects of public and private access, consider the following program:

```java
/* This program demonstrates the difference between public and private.
 */
class Test {
    int a; // default access
    public int b; // public access
    private int c; // private access

    // methods to access c
    void setc(int i) { // set c's value
        c = i;
    }
    int getc() { // get c's value
        return c;
    }
}
```

```java
class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();

        // These are OK, a and b may be accessed directly
        ob.a = 10;
        ob.b = 20;

        // This is not OK and will cause an error
        // ob.c = 100; // Error!
    }
}
```
As you can see, inside the Test class, a uses default access, which for this example is the same as specifying public. b is explicitly specified as public. Member c is given private access. This means that it cannot be accessed by code outside of its class. So, inside the AccessTest class, c cannot be used directly. It must be accessed through its public methods: setc() and getc(). If you were to remove the comment symbol from the beginning of the following line,

```java
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the Stack class shown at the end of Chapter 6.

```java
// This class defines an integer stack that can hold 10 values.
class Stack {
    /* Now, both stck and tos are private. This means
       that they cannot be accidentally or maliciously
       altered in a way that would be harmful to the stack. */
    private int stck[] = new int[10];
    private int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
```
if(tos==9)  
    System.out.println("Stack is full.");  
else  
    stck[++tos] = item;  
}  
// Pop an item from the stack  
int pop() {  
    if(tos < 0) {  
        System.out.println("Stack underflow.");  
        return 0;  
    }  
    else  
        return stck[tos--];  
}  

As you can see, now both stck, which holds the stack, and tos, which is the index of the top of the stack, are specified as private. This means that they cannot be accessed or altered except through push( ) and pop( ). Making tos private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the stck array.

The following program demonstrates the improved Stack class. Try removing the commented-out lines to prove to yourself that the stck and tos members are, indeed, inaccessible.

class TestStack {  
    public static void main(String args[]) {  
        Stack mystack1 = new Stack();  
        Stack mystack2 = new Stack();  

        // push some numbers onto the stack  
        for(int i=0; i<10; i++) mystack1.push(i);  
        for(int i=10; i<20; i++) mystack2.push(i);  

        // pop those numbers off the stack  
        System.out.println("Stack in mystack1:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack1.pop());  
        System.out.println("Stack in mystack2:");  
    }  
}
for(int i=0; i<10; i++)
    System.out.println(mystack2.pop());

    // these statements are not legal
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
}
}

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is `main()`. `main()`
is declared as static because it must be called before any objects exist.

Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables, and a static initialization block:

```java
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) { 
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String args[]) { 
        meth(42);
    }
}
```
As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Here is the output of the program:

Static block initialized.
x = 42
a = 3
b = 12

Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a static method from outside its class, you can do so using the following general form:

classname.method( )

Here, classname is the name of the class in which the static method is declared. As you can see, this format is similar to that used to call non-static methods through object-reference variables. A static variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside main( ), the static method
callme( ) and the static variable b are accessed through their class name StaticDemo.

class StaticDemo {
    static int a = 42;
    static int b = 99;

    static void callme() {
        System.out.println("a = " + a);
    }
}

class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}

Here is the output of this program:

    a = 42
    b = 99

Introducing final
A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

    final int FILE_NEW = 1;
    final int FILE_OPEN = 2;
    final int FILE_SAVE = 3;
    final int FILE_SAVEAS = 4;
    final int FILE_QUIT = 5;

Subsequent parts of your program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed. It is a common coding
convention to choose all uppercase identifiers for final fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared final. Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once.

The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This additional usage of final is described in the next chapter, when inheritance is described.

Arrays Revisited
Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its length instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```java
// This program demonstrates the length array member.
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is "+ a1.length);
    System.out.println("length of a2 is "+ a2.length);
    System.out.println("length of a3 is "+ a3.length);
  }
}
```
This program displays the following output:

length of a1 is 10
length of a2 is 8
length of a3 is 4

As you can see, the size of each array is displayed. Keep in mind that the value of length has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the length member to good use in many situations. For example, here is an improved version of the Stack class. As you might recall, the earlier versions of this class always created a ten-element stack. The following version lets you create stacks of any size. The value of stck.length is used to prevent the stack from overflowing.
Improved Stack class that uses the length array member.

class Stack {
    private int stk[];
    private int tos;

    // allocate and initialize stack
    Stack(int size) {
        stk = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==stk.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stk[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stk[tos--];
    }
}

class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

Introducing Nested and Inner Classes
It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and
use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner.

```java
// Demonstrate an inner class.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x. An instance method named display( ) is defined inside Inner. This method displays outer_x on the standard output stream. The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( ) method. That method creates an instance of class Inner and the display( ) method is called.

It is important to realize that an instance of Inner can
be created only within the scope of class Outer. The Java compiler generates an error message if any code outside of class Outer attempts to instantiate class Inner. In general, an inner class instance must be created by an enclosing scope.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```java
// This program will not compile.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        int y = 10; // y is local to Inner

        void display() {
            System.out.println("display: outer_x = "+ outer_x);
        }
    }

    void showy() {
        System.out.println(y); // error, y not known here!
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Here, y is declared as an instance variable of Inner. Thus, it is not known outside of that class and it cannot
Although we have been focusing on inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a for loop, as this next program shows:

```java
// Define an inner class within a for loop.
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

The output from this version of the program is shown here:

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```
While nested classes are not applicable to all situations, they are particularly helpful when handling events. We will return to the topic of nested classes in Chapter 22. There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about anonymous inner classes, which are inner classes that don’t have a name.

One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

Exploring the String Class
Although the String class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the example programs shown toward the end of Part I. String is probably the most commonly used class in Java’s class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type String. Even string constants are actually String objects. For example, in the statement

```java
System.out.println("This is a String, too");
```

the string “This is a String, too” is a String object.

The second thing to understand about strings is that objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:
• If you need to change a string, you can always create a new one that contains the modifications.

• Java defines a peer class of String, called StringBuffer, which allows strings to be altered, so all of the normal string manipulations are still available in Java. (StringBuffer is described in Part II of this book.)

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

```java
String myString = "this is a test";
```

Once you have created a String object, you can use it anywhere that a string is allowed. For example, this statement displays myString:

```java
System.out.println(myString);
```

Java defines one operator for String objects: +. It is used to concatenate two strings. For example, this statement

```java
String myString = "I" + " like " + "Java."
```

results in myString containing “I like Java.”

The following program demonstrates the preceding concepts:

```java
// Demonstrating Strings.
class StringDemo {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1 + " and " + strOb2;

        System.out.println(strOb1);
        System.out.println(strOb2);
        System.out.println(strOb3);
    }
}
```
The output produced by this program is shown here:

First String
Second String
First String and Second String

The String class contains several methods that you can use. Here are a few. You can test two strings for equality by using equals( ). You can obtain the length of a string by calling the length( ) method. You can obtain the character at a specified index within a string by calling charAt( ). The general forms of these three methods are shown here:

boolean equals(secondStr)
int length()
char charAt(index)

Here is a program that demonstrates these methods:

// Demonstrating some String methods.
class StringDemo2 {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;

        System.out.println("Length of strOb1: " + strOb1.length());
        System.out.println("Char at index 3 in strOb1: " + strOb1.charAt(3));

        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");

        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
    }
}
else
    System.out.println("strOb1 != strOb3");
}

This program generates the following output:

Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

// Demonstrate String arrays.
class StringDemo3 {
    public static void main(String args[]) {
        String str[] = { "one", "two", "three" };

        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " + str[i]);
    }
}

Here is the output from this program:

str[0]: one
str[1]: two
str[2]: three

As you will see in the following section, string arrays play an important part in many Java programs.

Using Command-Line Arguments
Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ). A command-line argument is the information that directly follows the program’s name on the command line when it is executed. To access the command-line arguments
inside a Java program is quite easy—they are stored as strings in a String array passed to the args parameter of main( ). The first command-line argument is stored at args[0], the second at args[1], and so on. For example, the following program displays all of the command-line arguments that it is called with:

```java
// Display all command-line arguments.
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
            System.out.println("args[" + i + "]: " + args[i]);
    }
}

Try executing this program, as shown here:
java CommandLine this is a test 100 -1

When you do, you will see the following output:

args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1

REMEMBER
All command line arguments are passed as strings.
You must convert numeric values to their internal forms manually, as explained in Chapter 16.

Varargs: Variable-Length Arguments
Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called varargs and it is short for variable-length arguments. A
method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.

Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. Another example is the printf( ) method that is part of Java’s I/O library. As you will see in Chapter 19, it takes a variable number of arguments, which it formats and then outputs.

Prior to JDK 5, variable-length arguments could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some cases, it applies to only a narrow class of situations.

In cases where the maximum number of potential arguments was larger, or unknowable, a second approach was used in which the arguments were put into an array, and then the array was passed to the method. This approach is illustrated by the following program:

```java
// Use an array to pass a variable number of arguments to a method. This is the old-style approach to variable-length arguments.
```
class PassArray {
    static void vaTest(int v[]) {
        System.out.print("Number of args: "+v.length +
        " Contents: ");

        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        // Notice how an array must be created to
        // hold the arguments.
        int n1[] = { 10 };
        int n2[] = { 1, 2, 3 };
        int n3[] = { };

        vaTest(n1); // 1 arg
        vaTest(n2); // 3 args
        vaTest(n3); // no args
    }
}

The output from the program is shown here:

Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:

In the program, the method vaTest( ) is passed its arguments through the array v. This old-style approach to variable-length arguments does enable vaTest( ) to take an arbitrary number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling vaTest( ). Not only is it tedious to construct an array each time vaTest( ) is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods (...). For example, here is how vaTest( ) is written using a vararg:
static void vaTest(int ... v) {

This syntax tells the compiler that `vaTest()` can be called with zero or more arguments. As a result, `v` is implicitly declared as an array of type `int[]`. Thus, inside `vaTest()`, `v` is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

```java
// Demonstrate variable-length arguments.
class VarArgs {

    // vaTest() now uses a vararg.
    static void vaTest(int ... v) {
        System.out.print("Number of args: " + v.length + " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        // Notice how vaTest() can be called with a
        // variable number of arguments.
        vaTest(10); // 1 arg
        vaTest(1, 2, 3); // 3 args
        vaTest(); // no args
    }
}
```

The output from the program is the same as the original version.

There are two important things to notice about this program. First, as explained, inside `vaTest()`, `v` is operated on as an array. This is because `v` is an array. The `...` syntax simply tells the compiler that a variable number of arguments will be used, and that these arguments will be stored in the array referred to by `v`. Second, in `main()`, `vaTest()` is called with different
numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to \( v \). In the case of no arguments, the length of the array is zero.

A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

```java
int doIt(int a, int b, double c, int ... vals) {
```

In this case, the first three arguments used in a call to `doIt()` are matched to the first three parameters. Then, any remaining arguments are assumed to belong to `vals`.

Remember, the varargs parameter must be last. For example, the following declaration is incorrect:

```java
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!
```

Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal.

There is one more restriction to be aware of: there must be only one varargs parameter. For example, this declaration is also invalid:

```java
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!
```

The attempt to declare the second varargs parameter is illegal.

Here is a reworked version of the `vaTest()` method that takes a regular argument and a variable-length argument:

```java
// Use varargs with standard arguments.
```
class VarArgs2 {

    // Here, msg is a normal parameter and v is a
    // varargs parameter.
    static void vaTest(String msg, int … v) {
        System.out.print(msg + v.length +
                          " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[]) {
        vaTest("One vararg: ", 10);
        vaTest("Three varargs: ", 1, 2, 3);
        vaTest("No varargs: ");
    }
}

The output from this program is shown here:

    One vararg: 1 Contents: 10
    Three varargs: 3 Contents: 1 2 3
    No varargs: 0 Contents:

Overloading Vararg Methods

You can overload a method that takes a variable-length argument. For example, the following program overloads vaTest( ) three times:
The output produced by this program is shown here:

vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false

This program illustrates both ways that a varargs method can be overloaded. First, the types of its vararg parameter can differ. This is the case for vaTest(int ...).
and vaTest(boolean ...). Remember, the ... causes the parameter to be treated as an array of the specified type. Therefore, just as you can overload methods by using different types of array parameters, you can overload vararg methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with `vaTest(String, int ...)`. In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

**NOTE**

A varargs method can also be overloaded by a non-varargs method. For example, `vaTest(int x)` is a valid overload of `vaTest()` in the foregoing program. This version is invoked only when one int argument is present. When two or more int arguments are passed, the varargs version `vaTest(int...v)` is used.

**Varargs and Ambiguity**

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:
In this program, the overloading of `vaTest()` is perfectly correct. However, this program will not compile because of the following call:

```java
vaTest(); // Error: Ambiguous!
```

Because the vararg parameter can be empty, this call could be translated into a call to `vaTest(int ...)` or `vaTest(boolean ...)`. Both are equally valid. Thus, the call is inherently ambiguous.
Here is another example of ambiguity. The following overloaded versions of `vaTest( )` are inherently ambiguous even though one takes a normal parameter:

```java
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...
```

Although the parameter lists of `vaTest( )` differ, there is no way for the compiler to resolve the following call:

```java
vaTest(1)
```

Does this translate into a call to `vaTest(int ... )`, with one varargs argument, or into a call to `vaTest(int, int ... )` with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution.
Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

Inheritance Basics
To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. To see how, let's begin with a short example. The following program creates a superclass called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.
// A simple example of inheritance.

// Create a superclass.
class A {
    int i, j;

    void showij() {
        System.out.println("i and j: "+i+" "+j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    void showk() {
        System.out.println("k: "+k);
    }

    void sum() {
        System.out.println("i+j+k: "+(i+j+k));
    }
}
class SimpleInheritance {
    public static void main(String args []) {
        A superOb = new A();
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        /* The subclass has access to all public members of
           its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Sum of i, j and k in subOb: ");
        subOb.sum();
    }
}

The output from this program is shown here:

Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24

As you can see, the subclass B includes all of the members of its superclass, A. This is why subOb can access i and j and call showij( ). Also, inside sum( ), i and j can be referred to directly, as if they were part of B.
Even though A is a superclass for B, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a class declaration that inherits a superclass is shown here:

```java
class subclass-name extends superclass-name {
    // body of class
}
```

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

Member Access and Inheritance
Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private. For example, consider the following simple class hierarchy:
This program will not compile because the use of \texttt{j} inside the \texttt{sum()} method of \texttt{B} causes an access violation. Since \texttt{j} is declared as \texttt{private}, it is only accessible by other members of its own class. Subclasses have no access to it.
REMEMBER

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

A More Practical Example

Let’s look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the Box class developed in the preceding chapter will be extended to include a fourth component called weight. Thus, the new class will contain a box’s width, height, depth, and weight.
// This program uses inheritance to extend Box.
class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
class BoxWeight extends Box {
    double weight; // weight of box

    // constructor for BoxWeight
    BoxWeight(double w, double h, double d, double m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}

class DemoBoxWeight {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
    }
}

The output from this program is shown here:

Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

BoxWeight inherits all of the characteristics of Box and adds to them the weight component. It is not necessary for BoxWeight to re-create all of the features found in Box. It can simply extend Box to meet its own purposes.

A major advantage of inheritance is that once you
have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification. For example, the following class inherits Box and adds a color attribute:

```java
// Here, Box is extended to include color.
class ColorBox extends Box {
    int color; // color of box

    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}
```

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own unique attributes. This is the essence of inheritance.

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```java
class RefDemo {
    public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;
```
Here, `weightbox` is a reference to `BoxWeight` objects, and `plainbox` is a reference to `Box` objects. Since `BoxWeight` is a subclass of `Box`, it is permissible to assign `plainbox` a reference to the `weightbox` object.

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why `plainbox` can’t access `weight` even when it refers to a `BoxWeight` object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a `Box` reference to access the `weight` field, because `Box` does not define one.
Although the preceding may seem a bit esoteric, it has some important practical applications—two of which are discussed later in this chapter.

Using super

In the preceding examples, classes derived from Box were not implemented as efficiently or as robustly as they could have been. For example, the constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

super has two general forms. The first calls the superclass’ constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass
by use of the following form of super:

    super(arg-list);

Here, arg-list specifies any arguments needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass’ constructor.

To see how super( ) is used, consider this improved version of the BoxWeight class:

    // BoxWeight now uses super to initialize its Box attributes.
    class BoxWeight extends Box {
        double weight; // weight of box

        // initialize width, height, and depth using super()
        BoxWeight(double w, double h, double d, double m) {
            super(w, h, d); // call superclass constructor
            weight = m;
        }
    }

Here, BoxWeight( ) calls super( ) with the arguments w, h, and d. This causes the Box constructor to be called, which initializes width, height, and depth using these values. BoxWeight no longer initializes these values itself. It only needs to initialize the value unique to it: weight. This leaves Box free to make these values private if desired.

In the preceding example, super( ) was called with three arguments. Since constructors can be overloaded, super( ) can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of BoxWeight that provides constructors
for the various ways that a box can be constructed. In each case, super( ) is called using the appropriate arguments. Notice that width, height, and depth have been made private within Box.

```cpp
// A complete implementation of BoxWeight.
class Box {
    private double width;
    private double height;
    private double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

```
// compute and return volume
double volume() {
    return width * height * depth;
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }

    // constructor when all parameters are specified
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }

    // default constructor
    BoxWeight() {
        super();
        weight = -1;
    }
}
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}

DemoSuper {
public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    BoxWeight mybox3 = new BoxWeight(); // default
    BoxWeight mycube = new BoxWeight(3, 2);
    BoxWeight myclone = new BoxWeight(mybox1);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
    System.out.println();

    vol = mybox3.volume();
    System.out.println("Volume of mybox3 is " + vol);
    System.out.println("Weight of mybox3 is " + mybox3.weight);
    System.out.println();

    vol = myclone.volume();
    System.out.println("Volume of myclone is " + vol);
    System.out.println("Weight of myclone is " + myclone.weight);
    System.out.println();

    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    System.out.println("Weight of mycube is " + mycube.weight);
    System.out.println();
}
}
Pay special attention to this constructor in BoxWeight:

```java
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
}
```

Notice that `super()` is passed an object of type `BoxWeight`—not of type `Box`. This still invokes the constructor `Box(Box ob)`. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a `BoxWeight` object to the `Box` constructor. Of course, `Box` only has knowledge of its own members.

Let’s review the key concepts behind `super()`. When a subclass calls `super()`, it is calling the constructor of its immediate superclass. Thus, `super()` always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, `super()` must
always be the first statement executed inside a subclass constructor.

A Second Use for super

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

    super. member

Here, member can be either a method or an instance variable.

This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```java
// Using super to overcome name hiding.
class A {
    int i;
}

// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the i in A

    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }

    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}
```
class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);
        subOb.show();
    }
}

This program displays the following:

i in superclass: 1
i in subclass: 2

Although the instance variable i in B hides the i in A, super allows access to the i defined in the superclass. As you will see, super can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy
Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass BoxWeight is used as a superclass to create the subclass called Shipment. Shipment inherits all of the traits of BoxWeight and Box,
and adds a field called cost, which holds the cost of shipping such a parcel.

// Extend BoxWeight to include shipping costs.

// Start with Box.
class Box {
    private double width;
    private double height;
    private double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
class BoxWeight extends Box {
  double weight; // weight of box

  // construct clone of an object
  BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
  }

  // constructor when all parameters are specified
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // call superclass constructor
    weight = m;
  }

  // default constructor
  BoxWeight() {
    super();
    weight = -1;
  }

  // constructor used when cube is created
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}

// Add shipping costs.
class Shipment extends BoxWeight {
  double cost;

  // construct clone of an object
  Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
  }
}
The output of this program is shown here:

Volume of shipment1 is 3000.0
Weight of shipment 1 is 10.0
Shipping cost: $3.41

Volume of shipment 2 is 24.0
Weight of shipment 2 is 0.76
Shipping cost: $1.28

Because of inheritance, Shipment can make use of the previously defined classes of Box and BoxWeight, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: super() always refers to the constructor in the closest superclass. The super() in Shipment calls the constructor in BoxWeight. The super() in BoxWeight calls the constructor in Box. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a subclass needs parameters of its own.

NOTE
In the preceding program, the entire class hierarchy, including Box, BoxWeight, and Shipment, is shown all in one file. This is for your convenience only. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.

When Constructors Are Called
When a class hierarchy is created, in what order are the
When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called B and a superclass called A, is A’s constructor called before B’s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super() must be the first statement executed in a subclass’ constructor, this order is the same whether or not super() is used. If super( ) is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```java
// Demonstrate when constructors are called.

// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}
// Create a subclass by extending class A.
class B extends A {
    B() {
        System.out.println("Inside B's constructor.");
    }
}
// Create another subclass by extending B.
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}

class CallingCons {
    public static void main(String args[]) {
        C c = new C();
    }
}```
The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

As you can see, the constructors are called in order of derivation.

If you think about it, it makes sense that constructors are executed in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Method Overriding
In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:
The output produced by this program is shown here:

```
3
```

When `show()` is invoked on an object of type `B`, the version of `show()` defined within `B` is used. That is, the version of `show()` inside `B` overrides the version declared in `A`.

If you wish to access the superclass version of an
 overridden method, you can do so by using super. For example, in this version of B, the superclass version of show( ) is invoked within the subclass’ version. This allows all instance variables to be displayed.

```java
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    void show() {
        super.show(); // this calls A's show()
        System.out.println("k: " + k);
    }
}
```

If you substitute this version of A into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, super.show( ) calls the superclass version of show( ).

Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```java
// Methods with differing type signatures are overloaded – not overridden.
class A {
    int i, j;
```
```java
A(int a, int b) {
    i = a;
    j = b;
}

// display i and j
void show() {
    System.out.println("i and j: " + i + " " + j);
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // overload show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

The output produced by this program is shown here:

This is k: 3
i and j: 1 2

The version of `show( )` in B takes a string parameter. This makes its type signature different from the one in A, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of `show( )`
Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java’s most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let’s begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will
be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:
The output from the program is shown here:

Inside A's callme method
Inside B's callme method
Inside C's callme method

This program creates one superclass called A and two
subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A’s callme( ) method.

NOTE

Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

Why Overridden Methods?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Part of the key to successfully applying polymorphism
is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding
Let’s look at a more practical example that uses method overriding. The following program creates a superclass called Figure that stores the dimensions of a two-dimensional object. It also defines a method called area() that computes the area of an object. The program derives two subclasses from Figure. The first is Rectangle and the second is Triangle. Each of these subclasses overrides area() so that it returns the area of a rectangle and a triangle, respectively.
// Using run-time polymorphism.
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;

        figref = r;
        System.out.println("Area is " + figref.area());

        figref = t;
        System.out.println("Area is " + figref.area());

        figref = f;
        System.out.println("Area is " + figref.area());
    }
}

The output from the program is shown here:

Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0

Through the dual mechanisms of inheritance and runtime polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is
derived from Figure, then its area can be obtained by calling \texttt{area()}\texttt{). The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classes
There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class \texttt{Figure} used in the preceding example. The definition of \texttt{area()}\texttt{) is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class \texttt{Triangle}. It has no
meaning if area() is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java’s solution to this problem is the abstract method.

You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

    abstract type name(parameter-list);

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

Here is a simple example of a class with an abstract
method, followed by a class which implements that method:

```java
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();

    // concrete methods are still allowed in abstract classes
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}

class B extends A {
    void callme() {
        System.out.println("B's implementation of callme.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class A implements a concrete method called callmetoo(). This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java’s approach to run-time polymorphism is implemented through the use of superclass references.
Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the Figure class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares area() as abstract inside Figure. This, of course, means that all classes derived from Figure must override area().

```java
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // area is now an abstract method
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```
As the comment inside main( ) indicates, it is no longer possible to declare objects of type Figure, since it is now abstract. And, all subclasses of Figure must override area( ). To prove this to yourself, try creating a subclass that does not override area( ). You will receive a compile-time error.

Although it is not possible to create an object of type Figure, you can create a reference variable of type Figure. The variable figref is declared as a reference to Figure, which means that it can be used to refer to an object of any class derived from Figure. As explained, it is through superclass reference variables that overridden
methods are resolved at run time.

Using final with Inheritance

The keyword final has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of final apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java’s most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

```java
class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}
class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Because `meth( )` is declared as final, it cannot be overridden in `B`. If you attempt to do so, a compile-time error will result.

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be
overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with final methods. Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```java
final class A {
    //…
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    //…
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as final.
The Object Class
There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.

Object defines the following methods, which means that they are available in every object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object clone( )</td>
<td>Creates a new object that is the same as the object being cloned.</td>
</tr>
<tr>
<td>boolean equals(Object object)</td>
<td>Determines whether one object is equal to another.</td>
</tr>
<tr>
<td>void finalize( )</td>
<td>Called before an unused object is recycled.</td>
</tr>
<tr>
<td>Class&lt;?&gt; getClass( )</td>
<td>Obtains the class of an object at run time.</td>
</tr>
<tr>
<td>int hashCode( )</td>
<td>Returns the hash code associated with the invoking object.</td>
</tr>
<tr>
<td>void notify( )</td>
<td>Resumes execution of a thread waiting on the invoking object.</td>
</tr>
<tr>
<td>void notifyAll( )</td>
<td>Resumes execution of all threads waiting on the invoking object.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns a string that describes the object.</td>
</tr>
<tr>
<td>void wait( )</td>
<td>Waits on another thread of execution.</td>
</tr>
<tr>
<td>void wait(long milliseconds)</td>
<td></td>
</tr>
<tr>
<td>void wait(long milliseconds, int nanoseconds)</td>
<td></td>
</tr>
</tbody>
</table>

The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: equals( ) and toString( ). The equals( ) method compares two objects.
Returns true if the objects are equal, and false otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The toString() method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using println(). Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

One last point: Notice the unusual syntax in the return type for getClass(). This relates to Java’s generics feature, which is described in Chapter 14.
This chapter examines two of Java’s most innovative features: packages and interfaces. Packages are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

In previous chapters, you have seen how methods define the interface to the data in a class. Through the use of the interface keyword, Java allows you to fully abstract an interface from its implementation. Using interface, you can specify a set of methods that can be implemented by one or more classes. The interface, itself, does not actually define any implementation. Although they are similar to abstract classes, interfaces have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Packages
In the preceding chapters, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to
avoid name collisions. After a while, without some way
to manage the name space, you could run out of
convenient, descriptive names for individual classes. You
also need some way to be assured that the name you
choose for a class will be reasonably unique and not
collide with class names chosen by other programmers.
(Imagine a small group of programmers fighting over
who gets to use the name “Foobar” as a class name. Or,
imagine the entire Internet community arguing over who
first named a class “Espresso.”) Thankfully, Java provides
a mechanism for partitioning the class name space into
more manageable chunks. This mechanism is the
package. The package is both a naming and a visibility
control mechanism. You can define classes inside a
package that are not accessible by code outside that
package. You can also define class members that are
exposed only to other members of the same package.
This allows your classes to have intimate knowledge of
each other, but not expose that knowledge to the rest of
the world.

Defining a Package

To create a package is quite easy: simply include a
package command as the first statement in a Java source
file. Any classes declared within that file will belong to
the specified package. The package statement defines a
name space in which classes are stored. If you omit the
package statement, the class names are put into the
default package, which has no name. (This is why you haven’t had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the package statement:

```
package pkg;
```

Here, `pkg` is the name of the package. For example, the following statement creates a package called `MyPackage`:

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the `.class` files for any classes you declare to be part of `MyPackage` must be stored in a directory called `MyPackage`. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:
package pkg1[pkg2[pkg3]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable. Third, you can use the -classpath option with java and javac to specify the path to your classes.

For example, consider the following package specification:

package MyPack

In order for a program to find MyPack, one of three
things must be true. Either the program can be executed from a directory immediately above MyPack, or the CLASSPATH must be set to include the path to MyPack, or the -classpath option must specify the path to MyPack when the program is run via Java.

When the second two options are used, the class path must not include MyPack, itself. It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is

C:\MyPrograms\Java\MyPack

Then the class path to MyPack is

C:\MyPrograms\Java

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the .class files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:
Call this file AccountBalance.java and put it in a directory called MyPack.

Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory. Then, try executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above
MyPack when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path MyPack.)

As explained, AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

In the preceding chapters, you learned about various aspects of Java’s access control mechanism and its access modifiers. For example, you already know that access to a private member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java’s smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for
class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

While Java’s access control mechanism may seem complicated, we can simplify it as follows. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

<table>
<thead>
<tr>
<th></th>
<th>Private</th>
<th>No Modifier</th>
<th>Protected</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same class</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Same package subclass</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Same package non-subclass</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Different package subclass</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Different package non-subclass</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 9-1 Class Member Access
Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, p1 and p2.

The source for the first package defines three classes: Protection, Derived, and SamePackage. The first class defines four int variables in each of the legal protection modes. The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.
The second class, Derived, is a subclass of Protection in the same package, p1. This grants Derived access to every variable in Protection except for n_pri, the private one. The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri.

This is file Protection.java:

```java
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = "+ n);
        System.out.println("n_pri = "+ n_pri);
        System.out.println("n_pro = "+ n_pro);

        System.out.println("n_pub = "+ n_pub);
    }
}
```

This is file Derived.java:

```java
package p1;

class Derived extends Protection {
    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = "+ n);

        // class only
        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```
This is file SamePackage.java:

```java
package p1;

class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Following is the source code for the other package, p2. The two classes defined in p2 cover the other two conditions that are affected by access control. The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection’s variables except for n_pri (because it is private) and n, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class OtherPackage has access to only one variable, n_pub, which was declared public.

This is file Protection2.java:

```java
package p2;
```
class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("derived other package constructor");
    // class or package only
    // System.out.println("n = " + n);
    // class only
    // System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}

This is file OtherPackage.java:

package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

    // class or package only
    // System.out.println("n = " + p.n);
    // class only
    // System.out.println("n_pri = " + p.n_pri);
    // class, subclass or package only
    // System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}

If you want to try these two packages, here are two test files you can use. The one for package p1 is shown here:

// Demo package p1.
The test file for p2 is shown next:

// Demo package p2.
package p2;

// Instantiate the various classes in p2.
public class Demo {
    public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
    }
}

Importing Packages
Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be
referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing.

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement:

```
import pkg1 [.pkg2].(classname | *)
```

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (\.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang. Normally, you have to import every package or class that you want to use, but since
Java is useless without much of the functionality in java.lang, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```java
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

It must be emphasized that the import statement is optional. Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy. For example, this fragment uses an import statement:

```java
import java.util.*;
class MyDate extends Date {
}
```

The same example without the import statement looks like this:

```java
class MyDate extends java.util.Date {
}
```

In this version, Date is fully-qualified.

As shown in Table 9-1, when a package is imported, only those items within the package declared as public
will be available to non-subclasses in the importing code. For example, if you want the Balance class of the package MyPack shown earlier to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

```java
package MyPack;

/* Now, the Balance class, its constructor, and its show() method are public. This means that they can be used by non-subclass code outside their package. */

public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if (bal<0)
            System.out.print("-- > ");
        System.out.println(name + ": "+ bal);
    }
}

As you can see, the Balance class is now public. Also, its constructor and its show() method are public, too. This means that they can be accessed by any type of code outside the MyPack package. For example, here TestBalance imports MyPack and is then able to make use of the Balance class:

```
class TestBalance {
    public static void main(String args[]) {
        /* Because Balance is public, you may use Balance
        class and call its constructor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);
        test.show(); // you may also call show()
    }
}

As an experiment, remove the public specifier from
the Balance class and then try compiling TestBalance. As
explained, errors will result.

Interfaces
Using the keyword interface, you can fully abstract a
class’ interface from its implementation. That is, using
interface, you can specify what a class must do, but not
how it does it. Interfaces are syntactically similar to
classes, but they lack instance variables, and their
methods are declared without any body. In practice, this
means that you can define interfaces that don’t make
assumptions about how they are implemented. Once it is
defined, any number of classes can implement an
interface. Also, one class can implement any number of
interfaces.

To implement an interface, a class must create the
complete set of methods defined by the interface.
However, each class is free to determine the details of its
own implementation. By providing the interface
keyword, Java allows you to fully utilize the “one
Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

NOTE
Interfaces add most of the functionality that is required for many applications that would normally resort to using multiple inheritance in a language such as C++.

Defining an Interface
An interface is defined much like a class. This is a simplified general form of an interface:
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);

type final-varname1 = value;
    type final-varname2 = value;
    //...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. name is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly final and static,
meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly public.

Here is an example of an interface definition. It declares a simple interface that contains one method called `callback()` that takes a single integer parameter.

```java
interface Callback {
    void callback(int param);
}
```

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the `implements` clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the `implements` clause looks like this:

```java
class classname [extends superclass] [implements interface [,interface...]] {
    // class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared `public`. Also, the type signature of the
implementing method must match exactly the type signature specified in the interface definition.

Here is a small example class that implements the Callback interface shown earlier:

```java
class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("callback called with " + p);
    }
}
```

Notice that `callback( )` is declared using the public access modifier.

**REMEMBER**

When you implement an interface method, it must be declared as public.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of Client implements `callback( )` and adds the method `nonIfaceMeth( )`:

```java
class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("callback called with " + p);
    }

    void nonIfaceMeth() {
        System.out.println("Classes that implement interfaces " + "may also define other members, too.");
    }
}
```
Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee.” This process is similar to using a superclass reference to access a subclass object, as described in Chapter 8.

CAUTION

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the `callback()` method via an interface reference variable:

```java
class TestIface {
}```
public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
}

The output of this program is shown here:

callback called with 42

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback( ) method, it cannot access any other members of the Client class. An interface reference variable has knowledge only of the methods declared by its interface declaration. Thus, c could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, shown here:

    // Another implementation of Callback.
    class AnotherClient implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
            System.out.println("Another version of callback");
            System.out.println("p squared is " + (p*p));
        }
    }

Now, try the following class:
class TestIface2 {
    public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();

        c.callback(42);

        c = ob; // c now refers to AnotherClient object
        c.callback(42);
    }
}

The output from this program is shown here:

callback called with 42
Another version of callback
p squared is 1764

As you can see, the version of callback( ) that is called is determined by the type of object that c refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

Partial Implementations
If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract. For example:

abstract class Incomplete implements Callback {
    int a, b;

    void show() {
        System.out.println(a + " " + b);
    }
    //...
}

Here, the class Incomplete does not implement callback(}
and must be declared as abstract. Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface. A nested interface can be declared as public, private, or protected. This differs from a top-level interface, which must either be declared as public or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:
Notice that A defines a member interface called NestedIF and that it is declared public. Next, B implements the nested interface by specifying

    implements A.NestedIF

Notice that the name is fully qualified by the enclosing class' name. Inside the main( ) method, an A.NestedIF reference called nif is created, and it is assigned a reference to a B object. Because B implements A.NestedIF, this is legal.
Applying Interfaces

To understand the power of interfaces, let’s look at a more practical example. In earlier chapters, you developed a class called Stack that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be “growable.” The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods \texttt{push( )} and \texttt{pop( )} define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let’s look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called \texttt{IntStack.java}. This interface will be used by both stack implementations.

```java
// Define an integer stack interface.
interface IntStack {
    void push(int item); // store an item
    int pop(); // retrieve an item
}
```

The following program creates a class called \texttt{FixedStack} that implements a fixed-length version of an integer stack:
class FixedStack implements IntStack {
    private int stck[];
    private int tos;

    // allocate and initialize stack
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        if (tos == stck.length - 1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    public int pop() {
        if (tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.
// Implement a "growable" stack.
class DynStack implements IntStack {
    private int stck[];
    private int tos;

    // allocate and initialize stack
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        // if stack is full, allocate a larger stack
        if(tos==stck.length-1) {
            int temp[] = new int[stck.length * 2]; // double size
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        }
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    public int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
The following class uses both the FixedStack and DynStack implementations. It does so through an interface reference. This means that calls to push( ) and pop( ) are resolved at run time rather than at compile time.
In this program, mystack is a reference to the IntStack interface. Thus, when it refers to ds, it uses the versions of push( ) and pop( ) defined by the DynStack implementation. When it refers to fs, it uses the versions of push( ) and pop( ) defined by FixedStack. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces
You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations.) If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is as if that class were importing the constant fields into the class name space as final variables. The next example uses this technique to implement an automated “decision maker”: 
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;  // 30%
        else if (prob < 60)
            return YES;  // 30%
        else if (prob < 75)
            return LATER;  // 15%
        else if (prob < 98)
            return SOON;  // 13%
        else
            return NEVER;  // 2%
    }
}
class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();

        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}

Notice that this program makes use of one of Java's standard classes: Random. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method nextDouble( ) is used. It returns random numbers in the range 0.0 to 1.0.
In this sample program, the two classes, Question and AskMe, both implement the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

Later
Soon
No
Yes

Interfaces Can Be Extended

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:
As an experiment, you might want to try removing the implementation for `meth1()` in `MyClass`. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.
Although the examples we’ve included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.
This chapter examines Java’s exception-handling mechanism. An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java’s exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals
A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java runtime system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some
error condition to the caller of a method.

Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

This is the general form of an exception-handling block:
Here, ExceptionType is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

**NOTE**

JDK 7 adds a new form of the try statement that supports automatic resource management. This new form of try, called try-with-resources, is described in Chapter 13 in the context of managing files because files are some of the most commonly used resources.

**Exception Types**

All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception.
This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type Error, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

The top-level exception hierarchy is shown here:
Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don’t handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```java
class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven’t supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java:4)
```
Notice how the class name, Exc0; the method name, main; the filename, Exc0.java; and the line number, 4, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from main( ):

```java
class Exc1 {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Exc1.subroutine();
    }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
    at Exc1.subroutine(Exc1.java:4)
    at Exc1.main(Exc1.java:7)
```

As you can see, the bottom of the stack is main’s line 7, which is the call to subroutine( ), which caused the
exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch
Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

class Exc2 {
    public static void main(String args[]) {
        int d, a;

        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }
    }
}
This program generates the following output:

Division by zero.
After catch statement.

Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Put differently, catch is not “called,” so execution never “returns” to the try block from a catch. Thus, the line “This will not be printed.” is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try / catch mechanism.

A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement. A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described shortly). The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement.

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For
example, in the next program each iteration of the for loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into a. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

```java
// Handle an exception and move on.
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e) {
                System.out.println("Division by zero.");
                a = 0; // set a to zero and continue
            }
            System.out.println("a: " + a);
        }
    }
}
```

Displaying a Description of an Exception

`Throwable` overrides the `toString()` method (defined by `Object`) so that it returns a string containing a description of the exception. You can display this description in a `println()` statement by simply passing the exception as an argument. For example, the catch block in the preceding program can be rewritten like this:
catch (ArithmeticException e) {
    System.out.println("Exception: "+ e);
    a = 0; // set a to zero and continue
}

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch Clauses
In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try / catch block. The following example traps two different exception types:

// Demonstrate multiple catch statements.
class MultipleCatches {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            throw new ArrayIndexOutOfBoundsException("Index: ");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Element out of bounds.");
        } catch (ArithmeticException e) {
            System.out.println("Exception: "+ e);
        }
    }
}
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch (ArithmeticException e) {
    System.out.println("Divide by 0: " + e);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}

This program will cause a division-by-zero exception if it is started with no command-line arguments, since \( a \) will equal zero. It will survive the division if you provide a command-line argument, setting \( a \) to something larger than zero. But it will cause an ArrayIndexOutOfBoundsException, since the int array \( c \) has a length of 1, yet the program attempts to assign a value to \( c[42] \).

Here is the output generated by running it both ways:

C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.

When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a
subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```java
/* This program contains an error.
A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.
*/
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached because ArithmeticException is a subclass of Exception. */
        catch(ArithmeticException e) { // ERROR – unreachable
            System.out.println("This is never reached.");
        }
    }
}
```

If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException. This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

Nested try Statements
The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement’s catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested try statements:
As you can see, this program nests one try block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on...
to the outer try block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block. Here are sample runs that illustrate each case:

```java
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
   java.lang.ArrayIndexOutOfBoundsException:42
```

Nesting of try statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a try block. Inside that method is another try statement. In this case, the try within the method is still nested inside the outer try block, which calls the method. Here is the previous program recoded so that the nested try block is moved inside the method nesttry():

```java
```
Try statements can be implicitly nested via calls to methods. */
class MethNestTry {
    static void nesttry(int a) {
        try { // nested try block
            /* If one command-line arg is used,
                then a divide-by-zero exception
                will be generated by the following code. */
            if(a==1) a = a/(a-a); // division by zero

            /* If two command-line args are used,
                then generate an out-of-bounds exception. */
            if(a==2) {
                int c[] = { 1 };
                c[42] = 99; // generate an out-of-bounds exception
            }
            catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }
        }
    }
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command-line args are present,
                the following statement will generate
                a divide-by-zero exception. */
            int b = 42 / a;
            System.out.println("a = " + a);
            nesttry(a);
        } catch(ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}

The output of this program is identical to that of the preceding example.

throw
So far, you have only been catching exceptions that are
thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

```java
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```java
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
```
This program gets two chances to deal with the same error. First, `main()` sets up an exception context and then calls `demoproc()`. The `demoproc()` method then sets up another exception-handling context and immediately throws a new instance of `NullPointerException`, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

The program also illustrates how to create one of Java’s standard exception objects. Pay close attention to this line:

throw new NullPointerException("demo");

Here, `new` is used to construct an instance of `NullPointerException`. Many of Java’s built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When
the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a call to getMessage( ), which is defined by Throwable.

```java
throws
```

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method’s declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a throws clause:

```java
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.
Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```java
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

To make this example compile, you need to make two changes. First, you need to declare that `throwOne()` throws `IllegalAccessException`. Second, `main()` must define a try / catch statement that catches this exception.

The corrected example is shown here:

```java
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Here is the output generated by running this example program:
When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.

finally creates a block of code that will be executed after a try /catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.
Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

```java
// Demonstrate finally.
public class FinallyDemo {
    // Through an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }

    // Return from within a try block.
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }
}
```
In this example, procA() prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB()’s try statement is exited via a return statement. The finally clause is executed before procB() returns. In procC(), the try statement executes normally, without error. However, the finally block is still executed.

REMEMBER

If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

Here is the output generated by the preceding program:

inside procA
procA's finally
Exception caught
inside procB
Java’s Built-in Exceptions

Inside the standard package java.lang, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type RuntimeException. As previously explained, these exceptions need not be included in any method’s throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed in Table 10-1. Table 10-2 lists those exceptions defined by java.lang that must be included in a method’s throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. Java defines several other types of exceptions that relate to its various class libraries.
<table>
<thead>
<tr>
<th>Exception</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticException</td>
<td>Arithmetic error, such as divide-by-zero.</td>
</tr>
<tr>
<td>ArrayIndexOutOfBoundsException</td>
<td>Array index is out-of-bounds.</td>
</tr>
<tr>
<td>ArrayStoreException</td>
<td>Assignment to an array element of an incompatible type.</td>
</tr>
<tr>
<td>ClassCastException</td>
<td>Invalid cast.</td>
</tr>
<tr>
<td>EnumConstantNotPresentException</td>
<td>An attempt is made to use an undefined enumeration value.</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>Illegal argument used to invoke a method.</td>
</tr>
<tr>
<td>IllegalMonitorStateException</td>
<td>Illegal monitor operation, such as waiting on an unlocked thread.</td>
</tr>
<tr>
<td>IllegalStateException</td>
<td>Environment or application is in incorrect state.</td>
</tr>
<tr>
<td>IllegalThreadStateException</td>
<td>Requested operation not compatible with current thread state.</td>
</tr>
<tr>
<td>IndexOutOfBoundsException</td>
<td>Some type of index is out-of-bounds.</td>
</tr>
<tr>
<td>NegativeArraySizeException</td>
<td>Array created with a negative size.</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>Invalid use of a null reference.</td>
</tr>
<tr>
<td>NumberFormatException</td>
<td>Invalid conversion of a string to a numeric format.</td>
</tr>
<tr>
<td>SecurityException</td>
<td>Attempt to violate security.</td>
</tr>
<tr>
<td>StringIndexOutOfBoundsException</td>
<td>Attempt to index outside the bounds of a string.</td>
</tr>
<tr>
<td>TypeNotFoundException</td>
<td>Type not found.</td>
</tr>
<tr>
<td>UnsupportedOperationException</td>
<td>An unsupported operation was encountered.</td>
</tr>
</tbody>
</table>

Table 10-1 Java’s Unchecked RuntimeException
Subclasses Defined in java.lang

<table>
<thead>
<tr>
<th>Exception</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassNotFoundException</td>
<td>Class not found.</td>
</tr>
<tr>
<td>CloneNotSupportedException</td>
<td>Attempt to clone an object that does not implement the Cloneable interface.</td>
</tr>
<tr>
<td>InstantiationException</td>
<td>Access to a class is denied.</td>
</tr>
<tr>
<td>InstantiationException</td>
<td>Attempt to create an object of an abstract class or interface.</td>
</tr>
<tr>
<td>InterruptedException</td>
<td>One thread has been interrupted by another thread.</td>
</tr>
<tr>
<td>NoSuchFieldException</td>
<td>A requested field does not exist.</td>
</tr>
<tr>
<td>NoSuchMethodException</td>
<td>A requested method does not exist.</td>
</tr>
<tr>
<td>ReflectiveOperationException</td>
<td>Superclass of reflection-related exceptions. (Added by JDK 7.)</td>
</tr>
</tbody>
</table>

Table 10-2 Java’s Checked Exceptions Defined in
Creating Your Own Exception Subclasses

Although Java’s built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable). Your subclasses don’t need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. They are shown in Table 10-3. You may also wish to override one or more of these methods in exception classes that you create.

Exception defines four constructors. Two support chained exceptions, described in the next section. The other two are shown here:

```java
Exception()
Exception(String msg)
```

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to
override toString( ). Here’s why: The version of toString( ) defined by Throwable (and inherited by Exception) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding toString( ), you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>final void addSuppressed( Throwable exc)</td>
<td>Adds exc to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the new try-with-resources statement. (Added by JDK 7.)</td>
</tr>
<tr>
<td>Throwable fillInStackTrace( )</td>
<td>Returns a Throwable object that contains a completed stack trace. This object can be rethrown.</td>
</tr>
<tr>
<td>Throwable getCause( )</td>
<td>Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.</td>
</tr>
<tr>
<td>String getLocalizedMessage( )</td>
<td>Returns a localized description of the exception.</td>
</tr>
<tr>
<td>String getMessage( )</td>
<td>Returns a description of the exception.</td>
</tr>
<tr>
<td>StackTraceElement[ ] getStackTrace( )</td>
<td>Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Throwable initCause(Throwable causeExc)</td>
<td>Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.</td>
</tr>
<tr>
<td>void printStackTrace()</td>
<td>Displays the stack trace.</td>
</tr>
<tr>
<td>void printStackTrace(PrintStream stream)</td>
<td>Sends the stack trace to the specified stream.</td>
</tr>
<tr>
<td>void printStackTrace(PrintWriter stream)</td>
<td>Sends the stack trace to the specified stream.</td>
</tr>
<tr>
<td>void setStackTrace(StackTraceElement elements[])</td>
<td>Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.</td>
</tr>
</tbody>
</table>

Table 10-3 The Methods Defined by Throwable

The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing a carefully tailored description of the exception to be displayed.
This example defines a subclass of Exception called MyException. This subclass is quite simple: It has only a constructor plus an overridden toString() method that displays the value of the exception. The ExceptionDemo class defines a method named compute() that throws a MyException object. The exception is thrown when compute()'s integer parameter is greater than 10. The main() method sets up an exception handler for MyException, then calls compute() with a legal value (less than 10) and an illegal one to show both paths.
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]

Chained Exceptions

Beginning with JDK 1.4, a feature was incorporated into the exception subsystem: chained exceptions. The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to Throwable. The constructors are shown here:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

In the first form, causeExc is the exception that causes the
current exception. That is, `causeExc` is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the `Error`, `Exception`, and `RuntimeException` classes.

The chained exception methods added to `Throwable` are `getCause()` and `initCause()`. These methods are shown in Table 10-3 and are repeated here for the sake of discussion.

```java
public Throwable getCause()
public Throwable initCause(Throwable causeExc)
```

The `getCause()` method returns the exception that underlies the current exception. If there is no underlying exception, `null` is returned. The `initCause()` method associates `causeExc` with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. Thus, you can call `initCause()` only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can’t set it again using `initCause()`. In general, `initCause()` is used to set a cause for legacy exception classes that don’t support the two additional constructors described earlier.

Here is an example that illustrates the mechanics of handling chained exceptions:
// Demonstrate exception chaining.
class ChainExcDemo {
    static void demoproc() {

        // create an exception
        NullPointerException e =
            new NullPointerException("top layer");

        // add a cause
        e.initCause(new ArithmeticException("cause"));

        throw e;
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            // display top level exception
            System.out.println("Caught: " + e);

            // display cause exception
            System.out.println("Original cause: " +
                e.getCause());
        }
    }
}

The output from the program is shown here:

Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause

In this example, the top-level exception is NullPointerException. To it is added a cause exception, ArithmeticException. When the exception is thrown out of demoproc(), it is caught by main(). There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling getCause().

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself,
have a cause. Be aware that overly long chains of exceptions may indicate poor design.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Three New JDK 7 Exception Features

JDK 7 adds three interesting and useful features to the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the try statement called try-with-resources, and is described in Chapter 13 when files are introduced. The second new feature is called multi-catch, and the third is sometimes referred to as final rethrow or more precise rethrow. These two features are described here.

The multi-catch feature allows two or more exceptions to be caught by the same catch clause. It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions. Instead of having to catch each exception type individually, now you can use a single catch clause to handle all of the exceptions without code duplication.

To use a multi-catch, separate each exception type in the catch clause with the OR operator. Each multi-catch parameter is implicitly final. (You can explicitly specify
final, if desired, but it is not necessary.) Because each multi-catch parameter is implicitly final, it can’t be assigned a new value.

Here is a catch statement that uses the multi-catch feature to catch both ArithmeticException and ArrayIndexOutOfBoundsException:

```java
    catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

The following program shows the multi-catch feature in action:

```java
    // Demonstrate JDK 7's multi-catch feature.
    class MultiCatch {
        public static void main(String args[]) {
            int a=10, b=0;
            int vals[] = { 1, 2, 3 };

            try {
                int result = a / b; // generate an ArithmeticException
                // vals[10] = 19; // generate an ArrayIndexOutOfBoundsException
                // This catch clause catches both exceptions.
            } catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
                System.out.println("Exception caught: "+ e);
            }

            System.out.println("After multi-catch.");
        }
    }
```

The program will generate an ArithmeticException when the division by zero is attempted. If you comment out the division statement and remove the comment symbol from the next line, an ArrayIndexOutOfBoundsException is generated. Both exceptions are caught by the single catch statement.
The more precise rethrow feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated try block throws, that are not handled by a preceding catch clause, and that are a subtype or supertype of the parameter. Although this capability might not be needed often, it is now available for use. For the more precise rethrow feature to be in force, the catch parameter must be either effectively final, which means that it must not be assigned a new value inside the catch block, or explicitly declared final.

Using Exceptions
Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program’s logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

One last point: Java’s exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.
Unlike some computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For many readers, process-based multitasking is the more familiar form. A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can
format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java’s control. However, multithreaded multitasking is.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked environment in which Java operates because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a
much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

If you have programmed for operating systems such as Windows, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient because many of the details are handled for you.

The Java Thread Model
The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns
with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java’s multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

As most readers know, over the past few years, multi-core systems have become commonplace. Of course, single-core systems are still in widespread use. It is important to understand that Java’s multithreading features work in both types of systems. In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads
do not actually run at the same time, but idle CPU time is utilized. However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

NOTE

JDK 7 adds the Fork/Join Framework, which provides a powerful means of creating multithreaded applications that automatically scale to make best use of multi-core environments. The Fork/Join Framework is part of Java’s support for parallel programming, which is the name commonly given to the techniques that optimize some types of algorithms for parallel execution in systems that have more than one CPU. For a discussion of the Fork/Join Framework and other concurrency utilities, see Chapter 27. Java’s traditional multithreading capabilities are described here.

Threads exist in several states. Here is a general description. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily halts its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once
Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn’t run any faster than a lower-priority thread if it is the only thread running. Instead, a thread’s priority is used to decide when to switch from one running thread to the next. This is called a context switch. The rules that determine when a context switch takes place are simple:

- A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

- A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows,
threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don’t, the other threads will not run.

CAUTION

Portability problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don’t conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the monitor. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.
Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class “Monitor”; instead, each object has its own implicit monitor that is automatically entered when one of the object’s synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging
After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java’s messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface
Java’s multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.
Since you can’t directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it. To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads. Several of those used in this chapter are shown here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>getName</td>
<td>Obtain a thread’s name.</td>
</tr>
<tr>
<td>getPriority</td>
<td>Obtain a thread’s priority.</td>
</tr>
<tr>
<td>isAlive</td>
<td>Determine if a thread is still running.</td>
</tr>
<tr>
<td>join</td>
<td>Wait for a thread to terminate.</td>
</tr>
<tr>
<td>run</td>
<td>Entry point for the thread.</td>
</tr>
<tr>
<td>sleep</td>
<td>Suspend a thread for a period of time.</td>
</tr>
<tr>
<td>start</td>
<td>Start a thread by calling its run method.</td>
</tr>
</tbody>
</table>

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use Thread and Runnable to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method `currentThread()`, which is a public static member of `Thread`. Its general form is shown here:

```
static Thread currentThread()
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let’s begin by reviewing the following example:

```java
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```
In this program, a reference to the current thread (the main thread, in this case) is obtained by calling `currentThread()`, and this reference is stored in the local variable `t`. Next, the program displays information about the thread. The program then calls `setName()` to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the `sleep()` method. The argument to `sleep()` specifies the delay period in milliseconds. Notice the try/catch block around this loop. The `sleep()` method in `Thread` might throw an `InterruptedException`. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when `t` is used as an argument to `println()`. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is `main`. Its priority is 5, which is the default value, and `main` is also the
name of the group of threads to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. After the name of the thread is changed, t is again output. This time, the new name of the thread is displayed.

Let’s look more closely at the methods defined by Thread that are used in the program. The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```java
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException.

The sleep( ) method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

```java
static void sleep(long milliseconds, int nanoseconds) throws InterruptedException
```

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using setName( ). You can obtain the name of a thread by calling getName( ) (but note
that this is not shown in the program). These methods are members of the Thread class and are declared like this:

    final void setName(String threadName)
    final String getName()

Here, threadName specifies the name of the thread.

Creating a Thread
In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

• You can implement the Runnable interface.

• You can extend the Thread class, itself.

The following two sections look at each method, in turn.

Implementing Runnable
The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

    public void run()

Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare
variables, just like the main thread can. The only difference is that run() establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run() returns.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName)

In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. In essence, start() executes a call to run(). The start() method is shown here:

void start()

Here is an example that creates a new thread and starts it running:
Inside NewThread’s constructor, a new Thread object is created by the following statement:

```java
// Create a second thread.
class NewThread implements Runnable {
    Thread t;

    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: "+t);
        t.start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: "+i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // create a new thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: "+i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

Inside NewThread’s constructor, a new Thread object is created by the following statement:

```
Passing this as the first argument indicates that you want the new thread to call the \texttt{run()} method on this object. Next, \texttt{start()} is called, which starts the thread of execution beginning at the \texttt{run()} method. This causes the child thread’s for loop to begin. After calling \texttt{start()}, \texttt{NewThread’s} constructor returns to \texttt{main()}. When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU in single-core systems, until their loops finish. The output produced by this program is as follows. (Your output may vary based upon the specific execution environment.)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang.” The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only
500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

Extending Thread

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread. Here is the preceding program rewritten to extend Thread:
This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of NewThread, which is derived from Thread.
Notice the call to super( ) inside NewThread. This invokes the following form of the Thread constructor:

```java
public Thread(String threadName)
```

Here, threadName specifies the name of the thread.

Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run( ). This is, of course, the same method required when you implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of Thread’s other methods, it is probably best simply to implement Runnable. Also, by implementing Runnable, your thread class does not need to inherit Thread, making it free to inherit a different class. Ultimately, which approach to use is up to you. However, throughout the rest of this chapter, we will create threads by using classes that implement Runnable.

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the
The following program creates three child threads:

```java
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: "+ t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");

        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution
As you can see, once started, all three child threads share the CPU. Notice the call to sleep(10000) in main(). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive() and join()

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling sleep() within main(), with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another
Thread has ended? Fortunately, Thread provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call isAlive( ) on the thread. This method is defined by Thread, and its general form is shown here:

```java
final boolean isAlive()
```

The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

While isAlive( ) is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called join( ), shown here:

```java
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses join( ) to ensure that the main thread is the last to stop. It also demonstrates the isAlive( ) method.
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

As you can see, after the calls to join( ) return, the threads have stopped executing.

Thread Priorities
Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a
lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run.

To set a thread’s priority, use the setPriority( ) method, which is a member of Thread. This is its general form:

    final void setPriority(int level)

Here, level specifies the new priority setting for the calling thread. The value of level must be within the
range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as static final variables within Thread.

You can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

```java
final int getPriority( )
```

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

**Synchronization**

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor. A monitor is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to
have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the synchronized keyword, and both are examined here.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object’s monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let’s begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, Callme, has a single method named call( ). The call( ) method takes a String parameter called msg. This method tries to print the msg string inside of square brackets. The interesting thing to notice is that after call( ) prints the opening bracket and the msg string, it calls
Thread.sleep(1000), which pauses the current thread for one second.

The constructor of the next class, Caller, takes a reference to an instance of the Callme class and a String, which are stored in target and msg, respectively. The constructor also creates a new thread that will call this object’s `run()` method. The thread is started immediately. The `run()` method of Caller calls the `call()` method on the target instance of Callme, passing in the msg string. Finally, the Synch class starts by creating a single instance of Callme, and three instances of Caller, each with a unique message string. The same instance of Callme is passed to each Caller.
// This program is not synchronized.

class Callme {
    void call(String msg) {
        System.out.print("[");
        System.out.print(msg);
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        target.call(msg);
    }
}

class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller obl1 = new Caller(target, "Hello");
        Caller obl2 = new Caller(target, "Synchronized");
        Caller obl3 = new Caller(target, "World");

        // wait for threads to end
        try {
            obl1.t.join();
            obl2.t.join();
            obl3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

Here is the output produced by this program:

Hello[Synchronized[World]
As you can see, by calling `sleep()` the `call()` method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a race condition, because the three threads are racing each other to complete the method. This example used `sleep()` to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can’t be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must serialize access to `call()`. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede `call()`’s definition with the keyword `synchronized`, as shown here:

```java
class Callme {
    synchronized void call(String msg) {
        ...
    }
}
```

This prevents other threads from entering `call()` while another thread is using it. After `synchronized` has been added to `call()`, the output of the program is as follows:

```
Hello
[Synchronized]
World
```

Any time that you have a method, or group of...
methods, that manipulates the internal state of an object in a multithreaded situation, you should use the synchronized keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can’t add synchronized to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a synchronized block.

This is the general form of the synchronized statement:

```java
synchronized(object) {
    // statements to be synchronized
```
Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object’s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the run( ) method:

```java
// This program uses a synchronized block.
class Callme {
    void call(String msg) {
        System.out.print('[' + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println(']');
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
```
Here, the call() method is not modified by synchronized. Instead, the synchronized statement is used inside Caller’s run() method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Interthread Communication
The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of
control through interprocess communication. As you will see, this is especially easy in Java.

As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the `wait()`, `notify()`, and `notifyAll()` methods. These methods are implemented as final methods in `Object`, so all classes have them. All three methods can be called only from within a synchronized context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- `wait()` tells the calling thread to give up the monitor
and go to sleep until some other thread enters the same monitor and calls notify( ).

- notify( ) wakes up a thread that called wait( ) on the same object.

- notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

These methods are declared within Object, as shown here:

```java
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Additional forms of wait( ) exist that allow you to specify a period of time to wait.

Before working through an example that illustrates interthread communication, an important point needs to be made. Although wait( ) normally waits until notify( ) or notifyAll( ) is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a spurious wakeup. In this case, a waiting thread resumes without notify( ) or notifyAll( ) having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, Oracle recommends that calls to wait( ) should take place within a loop that checks the condition on which the thread is waiting. The following example shows this
Let’s now work through an example that uses wait( ) and notify( ). To begin, consider the following sample program that incorrectly implements a simple form of the producer/ consumer problem. It consists of four classes: Q, the queue that you’re trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.
// An incorrect implementation of a producer and consumer.
class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}
Although the `put()` and `get()` methods on `Q` are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
```
As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use `wait()` and `notify()` to signal in both directions, as shown here:
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while (!valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        while (valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
            }
    }
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}
Inside `get()` method, `wait()` is called. This causes its execution to suspend until `Producer` notifies you that some data is ready. When this happens, execution inside `get()` resumes. After the data has been obtained, `get()` calls `notify()`. This tells `Producer` that it is okay to put more data in the queue. Inside `put()`, `wait()` suspends execution until `Consumer` has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and `notify()` is called. This tells `Consumer` that it should now remove it.

Here is some output from this program, which shows...
Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)
To understand deadlock fully, it is useful to see it in action. The next example creates two classes, A and B, with methods foo( ) and bar( ), respectively, which pause briefly before trying to call a method in the other class. The main class, named Deadlock, creates an A and a B instance, and then starts a second thread to set up the deadlock condition. The foo( ) and bar( ) methods use sleep( ) as a way to force the deadlock condition to occur.

```java
// An example of deadlock.
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("A Interrupted");
        }

        System.out.println(name + " trying to call B.last()");
        b.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}
```
class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("B Interrupted");
        }

        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
When you run this program, you will see the output shown here:

MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC. You will see that RacingThread owns the monitor on b, while it is waiting for the monitor on a. At the same time, MainThread owns a and is waiting to get b. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.
Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn’t want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2. Although you should use the modern approach for all new code, you still need to understand how these operations were accomplished for earlier Java environments. For example, you may need to update or maintain older, legacy code. You also need to understand why a change was made. For these reasons, the next section describes the original way that the execution of a thread was controlled, followed by a section that describes the modern approach.

Suspending, Resuming, and Stopping Threads Using Java 1.1 and Earlier

Prior to Java 2, a program used suspend( ) and resume( ), which are methods defined by Thread, to pause and restart the execution of a thread. They have the form shown below:

```java
final void suspend( )
```
Although these methods are no longer recommended, the following program demonstrates their use so that you can understand how they worked:

```java
// Using the suspend() and resume() methods for the purposes of demonstration only. Not for new code.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
    }
}
```
```java
    System.out.println(name + " exiting."
);} 
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread obl = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

        try {
            Thread.sleep(1000);
            obl.t.suspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            obl.t.resume();
            System.out.println("Resuming thread One");
            ob2.t.suspend();
            System.out.println("Suspending thread Two");

            Thread.sleep(1000);
            ob2.t.resume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }

        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish."");
            obl.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
        System.out.println("Main thread exiting.");
    }
} 
```
New thread: Thread[One, 5, main]
One: 15
New thread: Thread[Two, 5, main]
Two: 15
   One: 14
   Two: 14
   One: 13
   Two: 13
   One: 12
   Two: 12
   One: 11
   Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.

The Thread class also defines a method called stop() that stops a thread. Its signature is shown here:
final void stop( )

Once a thread has been stopped, it cannot be restarted using resume( ).

The Modern Way of Suspending, Resuming, and Stopping Threads

While the suspend( ), resume( ), and stop( ) methods defined by Thread seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The suspend( ) method of the Thread class was deprecated by Java 2 several years ago. This was done because suspend( ) can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

The resume( ) method is also deprecated. It does not cause problems, but cannot be used without the suspend( ) method as its counterpart.

The stop( ) method of the Thread class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data
structure might be left in a corrupted state. The trouble is that stop() causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Because you can’t now use the suspend(), resume(), or stop() methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the run() method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to “running,” the run() method must continue to let the thread execute. If this variable is set to “suspend,” the thread must pause. If it is set to “stop,” the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the wait() and notify() methods that are inherited from Object can be used to control the execution of a thread. This example is similar to the program in the previous section. However, the deprecated method calls have been removed. Let us consider the operation of this program.

The NewThread class contains a boolean instance variable named suspendFlag, which is used to control the execution of the thread. It is initialized to false by the
constructor. The run( ) method contains a synchronized statement block that checks suspendFlag. If that variable is true, the wait( ) method is invoked to suspend the execution of the thread. The mysuspend( ) method sets suspendFlag to true. The myresume( ) method sets suspendFlag to false and invokes notify( ) to wake up the thread. Finally, the main( ) method has been modified to invoke the mysuspend( ) and myresume( ) methods.
class NewThread implements Runnable {
    String name;  // name of thread
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start();  // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + " : " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }

        System.out.println(name + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }
}
synchronized void myResume() {
    suspendFlag = false;
    notify();
}

class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
“clean” as the old way, nevertheless, it is the way required to ensure that run-time errors don’t occur. It is the approach that must be used for all new code.

Obtaining A Thread’s State
As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the `getState()` method defined by `Thread`. It is shown here:

```java
Thread.State getState()
```

It returns a value of type `Thread.State` that indicates the state of the thread at the time at which the call was made. `State` is an enumeration defined by `Thread`. (An enumeration is a list of named constants. It is discussed in detail in Chapter 12.) Here are the values that can be returned by `getState()`:

<table>
<thead>
<tr>
<th>Value</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCKED</td>
<td>A thread that has suspended execution because it is waiting to acquire a lock.</td>
</tr>
<tr>
<td>NEW</td>
<td>A thread that has not begun execution.</td>
</tr>
<tr>
<td>RUNNABLE</td>
<td>A thread that either is currently executing or will execute when it gains access to the CPU.</td>
</tr>
<tr>
<td>TERMINATED</td>
<td>A thread that has completed execution.</td>
</tr>
<tr>
<td>TIMED_WAITING</td>
<td>A thread that has suspended execution for a specified period of time, such as when it has called <code>sleep()</code>. This state is also entered when a timeout version of <code>wait()</code> or <code>join()</code> is called.</td>
</tr>
<tr>
<td>WAITING</td>
<td>A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of <code>wait()</code> or <code>join()</code>.</td>
</tr>
</tbody>
</table>

*Figure 11-1* diagrams how the various thread states relate.
Given a Thread instance, you can use getState() to obtain the state of a thread. For example, the following sequence determines if a thread called thrd is in the RUNNABLE state at the time getState() is called:

```java
Thread.State ts = thrd.getState();
if(ts == Thread.State.RUNNABLE) // ...
```

It is important to understand that a thread’s state may change after the call to getState(). Thus, depending on the circumstances, the state obtained by calling getState() may not reflect the actual state of the thread only a moment later. For this (and other) reasons, getState() is not intended to provide a means of synchronizing threads. It’s primarily used for debugging or for profiling a thread’s run-time characteristics.
Using Multithreading
The key to utilizing Java’s multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember,
some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program! One last point: To create compute-intensive applications that can automatically scale to make use of the available processors in a multi-core system, consider using the new Fork/Join Framework, which is described in Chapter 27.
This chapter examines three relatively recent additions to the Java language: enumerations, autoboxing, and annotations (also referred to as metadata). Each expands the power of the language by offering a streamlined approach to handling common programming tasks. This chapter also discusses Java’s type wrappers and introduces reflection.

Enumerations
Versions of Java prior to JDK 5 lacked one feature that many programmers felt was needed: enumerations. In its simplest form, an enumeration is a list of named constants. Although Java offered other features that provide somewhat similar functionality, such as final variables, many programmers still missed the conceptual purity of enumerations—especially because enumerations are supported by most other commonly used languages. Beginning with JDK 5, enumerations were added to the Java language, and they are now available to the Java programmer.

In their simplest form, Java enumerations appear similar to enumerations in other languages. However, this similarity is only skin deep. In languages such as C++, enumerations are simply lists of named integer
constants. In Java, an enumeration defines a class type. By making enumerations into classes, the concept of the enumeration is greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Therefore, although enumerations were several years in the making, Java’s rich implementation made them well worth the wait.

Enumeration Fundamentals

An enumeration is created using the `enum` keyword. For example, here is a simple enumeration that lists various apple varieties:

```java
// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

The identifiers `Jonathan`, `GoldenDel`, and so on, are called enumeration constants. Each is implicitly declared as a public, static final member of `Apple`. Furthermore, their type is the type of the enumeration in which they are declared, which is `Apple` in this case. Thus, in the language of Java, these constants are called self-typed, in which “self” refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an `enum` using `new`. Instead, you declare and use an enumeration variable in much the same way as you do
one of the primitive types. For example, this declares `ap` as a variable of enumeration type `Apple`:

```java
Apple ap;
```

Because `ap` is of type `Apple`, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns `ap` the value `RedDel`:

```java
ap = Apple.RedDel;
```

Notice that the symbol `RedDel` is preceded by `Apple`.

Two enumeration constants can be compared for equality by using the `==` relational operator. For example, this statement compares the value in `ap` with the `GoldenDel` constant:

```java
if(ap == Apple.GoldenDel) // ...
```

An enumeration value can also be used to control a `switch` statement. Of course, all of the `case` statements must use constants from the same `enum` as that used by the `switch` expression. For example, this `switch` is perfectly valid:

```java
// Use an enum to control a switch statement.
switch(ap) {
    case Jonathan:
        // ...
        // ...
    case Winesap:
        // ...
```

Notice that in the `case` statements, the names of the
enumeration constants are used without being qualified by their enumeration type name. That is, Winesap, not Apple.Winesap, is used. This is because the type of the enumeration in the switch expression has already implicitly specified the enum type of the case constants. There is no need to qualify the constants in the case statements with their enum type name. In fact, attempting to do so will cause a compilation error.

When an enumeration constant is displayed, such as in a println() statement, its name is output. For example, given this statement:

    System.out.println(Apple.Winesap);

the name Winesap is displayed.

The following program puts together all of the pieces and demonstrates the Apple enumeration:

```java
// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo {
    public static void main(String args[])
    {
        Apple ap;

        ap = Apple.RedDel;

        // Output an enum value.
        System.out.println("Value of ap: " + ap);
        System.out.println();

        ap = Apple.GoldenDel;
```
// Compare two enum values.
if (ap == Apple.GoldenDel)
    System.out.println("ap contains GoldenDel.\n");

// Use an enum to control a switch statement.
switch (ap) {
    case Jonathan:
        System.out.println("Jonathan is red.");
        break;
    case GoldenDel:
        System.out.println("Golden Delicious is yellow.");
        break;
    case RedDel:
        System.out.println("Red Delicious is red.");
        break;
    case Winesap:
        System.out.println("Winesap is red.");
        break;
    case Cortland:
        System.out.println("Cortland is red.");
        break;
}
}

The output from the program is shown here:

Value of ap: RedDel

ap contains GoldenDel.

Golden Delicious is yellow.

The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: values( ) and valueOf( ). Their general forms are shown here:

    public static enum-type [ ] values( )
    public static enum-type valueOf(String str)
The `values()` method returns an array that contains a list of the enumeration constants. The `valueOf()` method returns the enumeration constant whose value corresponds to the string passed in `str`. In both cases, `enum-type` is the type of the enumeration. For example, in the case of the `Apple` enumeration shown earlier, the return type of `Apple.valueOf("Winesap")` is `Winesap`.

The following program demonstrates the `values()` and `valueOf()` methods:

```java
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String args[])
    {
        Apple ap;

        System.out.println("Here are all Apple constants:");

        // use values()
        Apple allapples[] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```

The output from the program is shown here:
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

Notice that this program uses a for-each style for loop to cycle through the array of constants obtained by calling values(). For the sake of illustration, the variable allapples was created and assigned a reference to the enumeration array. However, this step is not necessary because the for could have been written as shown here, eliminating the need for the allapples variable:

```java
for(Apple a : Apple.values())
    System.out.println(a);
```

Now, notice how the value corresponding to the name Winesap was obtained by calling valueOf().

```java
    ap = Apple.valueOf("Winesap");
```

As explained, valueOf() returns the enumeration value associated with the name of the constant represented as a string.

**NOTE**

C/C++ programmers will notice that Java makes it much easier to translate between the human-readable form of an enumeration constant and its binary value than do these other languages. This is a
significant advantage to Java’s approach to enumerations.

Java Enumerations Are Class Types

As explained, a Java enumeration is a class type. Although you don’t instantiate an enum using new, it otherwise has much the same capabilities as other classes. The fact that enum defines a class gives powers to the Java enumeration that enumerations in other languages simply do not have. For example, you can give them constructors, add instance variables and methods, and even implement interfaces.

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an enum, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration. For example, consider the following version of Apple:

```java
// Use an enum constructor, instance variable, and method.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

    private int price; // price of each apple

    // Constructor
    Apple(int p) { price = p; }

    int getPrice() { return price; }
}
```

```java
class EnumDemo3 {
```
public static void main(String args[]) {
    Apple ap;

    // Display price of Winesap.
    System.out.println("Winesap costs " +
            Apple.Winesap.getPrice() +
            " cents.\n");

    // Display all apples and prices.
    System.out.println("All apple prices:");
    for(Apple a : Apple.values())
        System.out.println(a + " costs " + a.getPrice() +
            " cents.");
}

The output is shown here:

Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.

This version of Apple adds three things. The first is the instance variable price, which is used to hold the price of each variety of apple. The second is the Apple constructor, which is passed the price of an apple. The third is the method getPrice(), which returns the value of price.

When the variable ap is declared in main(), the constructor for Apple is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside
parentheses after each constant, as shown here:

Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

These values are passed to the p parameter of Apple(), which then assigns this value to price. Again, the constructor is called once for each constant.

Because each enumeration constant has its own copy of price, you can obtain the price of a specified type of apple by calling getPrice(). For example, in main( ) the price of a Winesap is obtained by the following call:

Apple.Winesap.getPrice()

The prices of all varieties are obtained by cycling through the enumeration using a for loop. Because there is a copy of price for each enumeration constant, the value associated with one constant is separate and distinct from the value associated with another constant. This is a powerful concept, which is only available when enumerations are implemented as classes, as Java does.

Although the preceding example contains only one constructor, an enum can offer two or more overloaded forms, just as can any other class. For example, this version of Apple provides a default constructor that initializes the price to –1, to indicate that no price data is available:

// Use an enum constructor.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);
private int price;  // price of each apple

// Constructor
Apple(int p) { price = p; }

// Overloaded constructor
Apple() { price = -1; }

int getPrice() { return price; }
}

Notice that in this version, RedDel is not given an argument. This means that the default constructor is called, and RedDel’s price variable is given the value –1.

Here are two restrictions that apply to enumerations. First, an enumeration can’t inherit another class. Second, an enum cannot be a superclass. This means that an enum can’t be extended. Otherwise, enum acts much like any other class type. The key is to remember that each of the enumeration constants is an object of the class in which it is defined.

Enumerations Inherit(Enum)

Although you can’t inherit a superclass when declaring an enum, all enumerations automatically inherit one: java.lang.Enum. This class defines several methods that are available for use by all enumerations. The Enum class is described in detail in Part II, but three of its methods warrant a discussion at this time.

You can obtain a value that indicates an enumeration constant’s position in the list of constants. This is called its ordinal value, and it is retrieved by calling the
final int ordinal() method, shown here:

final int ordinal()
It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the Apple enumeration, Jonathan has an ordinal value of zero, GoldenDel has an ordinal value of 1, RedDel has an ordinal value of 2, and so on.

You can compare the ordinal value of two constants of the same enumeration by using the compareTo() method. It has this general form:

final int compareTo(enum-type e)
Here, enum-type is the type of the enumeration, and e is the constant being compared to the invoking constant. Remember, both the invoking constant and e must be of the same enumeration. If the invoking constant has an ordinal value less than e’s, then compareTo() returns a negative value. If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than e’s, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using equals(), which overrides the equals() method defined by Object. Although equals() can compare an enumeration constant to any other object, those two objects will be equal only
if they both refer to the same constant, within the same enumeration. Simply having ordinal values in common will not cause equals( ) to return true if the two constants are from different enumerations.

Remember, you can compare two enumeration references for equality by using = =. The following program demonstrates the ordinal( ), compareTo( ), and equals( ) methods:

```java
// Demonstrate ordinal(), compareTo(), and equals().

// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
    public static void main(String args[])
    {
        Apple ap, ap2, ap3;

        // Obtain all ordinal values using ordinal().
        System.out.println("Here are all apple constants and their ordinal values: ");
        for(Apple a : Apple.values())
            System.out.println(a + " " + a.ordinal());

        ap = Apple.RedDel;
        ap2 = Apple.GoldenDel;
        ap3 = Apple.RedDel;

        System.out.println();

        // Demonstrate compareTo() and equals()
        if(ap.compareTo(ap2) < 0)
            System.out.println(ap + " comes before " + ap2);

        if(ap.compareTo(ap2) > 0)
            System.out.println(ap2 + " comes before " + ap);
```
if (ap.compareTo(ap3) == 0)
    System.out.println(ap + " equals " + ap3);

System.out.println();

if (ap.equals(ap2))
    System.out.println("Error!");

if (ap.equals(ap3))
    System.out.println(ap + " equals " + ap3);

if (ap == ap3)
    System.out.println(ap + " == " + ap3);

}
}

The output from the program is shown here:

Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel

Another Enumeration Example

Before moving on, we will look at a different example that uses an enum. In Chapter 9, an automated “decision maker” program was created. In that version, variables called NO, YES, MAYBE, LATER, SOON, and NEVER were declared within an interface and used to represent the possible answers. While there is nothing technically
wrong with that approach, the enumeration is a better choice. Here is an improved version of that program that uses an enum called Answers to define the answers. You should compare this version to the original in Chapter 9.

```java
// An improved version of the "Decision Maker" program from Chapter 9. This version uses an enum, rather than interface variables, to represent the answers.

import java.util.Random;

// An enumeration of the possible answers.
enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random();
    Answers ask() {
        int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return Answers.MAYBE; // 15%
        else if (prob < 30)
            return Answers.NO; // 15%
        else if (prob < 60)
            return Answers.YES; // 30%
        else if (prob < 75)
            return Answers.LATER; // 15%
        else if (prob < 98)
            return Answers.SOON; // 13%
        else
            return Answers.NEVER; // 2%
    }
}
```
Type Wrappers
As you know, Java uses primitive types (also called simple types), such as int or double, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the
sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit Object.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can’t pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can’t use these data structures to store primitive types. To handle these (and other) situations, Java provides type wrappers, which are classes that encapsulate a primitive type within an object. The type wrapper classes are described in detail in Part II, but they are introduced here because they relate directly to Java’s autoboxing feature.

The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java’s object hierarchy. Each is briefly examined next.

Character
Character is a wrapper around a char. The constructor for Character is

    Character(char ch)
Here, ch specifies the character that will be wrapped by the Character object being created.

To obtain the char value contained in a Character object, call charValue( ), shown here:

```java
cchar charValue( )
```

It returns the encapsulated character.

Boolean

Boolean is a wrapper around boolean values. It defines these constructors:

```java
    Boolean(boolean boolValue)
    Boolean(String boolString)
```

In the first version, boolValue must be either true or false. In the second version, if boolString contains the string “true” (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

```java
    boolean booleanValue( )
```

It returns the boolean equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are Byte, Short,
Integer, Long, Float, and Double. All of the numeric type wrappers inherit the abstract class Number. Number declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```java
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

For example, `doubleValue( )` returns the value of an object as a double, `floatValue( )` returns the value as a float, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for `Integer`:

```java
Integer(int num)
Integer(String str)
```

If `str` does not contain a valid numeric value, then a `NumberFormatException` is thrown.

All of the type wrappers override `toString( )`. It returns the human-readable form of the value contained within
the wrapper. This allows you to output the value by passing a type wrapper object to println(), for example, without having to convert it into its primitive type.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```java
// Demonstrate a type wrapper.
class Wrap {
    public static void main(String args[]) {
        Integer iOb = new Integer(100);
        int i = iOb.intValue();
        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

This program wraps the integer value 100 inside an `Integer` object called `iOb`. The program then obtains this value by calling `intValue()` and stores the result in `i`.

The process of encapsulating a value within an object is called boxing. Thus, in the program, this line boxes the value 100 into an `Integer`:

```java
Integer iOb = new Integer(100);
```

The process of extracting a value from a type wrapper is called unboxing. For example, the program unboxes the value in `iOb` with this statement:

```java
int i = iOb.intValue();
```

The same general procedure used by the preceding
program to box and unbox values has been employed since the original version of Java. However, with the release of JDK 5, Java fundamentally improved on this through the addition of autoboxing, described next.

**Autoboxing**

Beginning with JDK 5, Java added two important features: autoboxing and auto-unboxing. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as `intValue()` or `doubleValue()`.

The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors. Moreover, it is very important to generics, which operate only on objects. Finally, autoboxing makes working with the Collections Framework (described in Part II) much easier.

With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct
an Integer object that has the value 100:

    Integer iOb = 100; // autobox an int

Notice that the object is not explicitly created through the use of new. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox iOb, you can use this line:

    int i = iOb; // auto-unbox

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

    // Demonstrate autoboxing/unboxing.
    class AutoBox {
        public static void main(String args[]) {
            Integer iOb = 100; // autobox an int

            int i = iOb; // auto-unbox

            System.out.println(i + " " + iOb); // displays 100 100
        }
    }

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
  // Take an Integer parameter and return
  // an int value;
  static int m(Integer v) {
    return v; // auto-unbox to int
  }

  public static void main(String args[]) {
    // Pass an int to m() and assign the return value
    // to an Integer. Here, the argument 100 is autoboxed
    // into an Integer. The return value is also autoboxed
    // into an Integer.
    Integer iOb = m(100);

    System.out.println(iOb);
  }
}

This program displays the following result:

100

In the program, notice that m( ) specifies an Integer parameter and returns an int result. Inside main( ), m( ) is passed the value 100. Because m( ) is expecting an Integer, this value is automatically boxed. Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed. Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.

Autoboxing/Unboxing Occurs in Expressions
In general, autoboxing and unboxing take place
whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```java
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
    public static void main(String args[]) {

        Integer iOb, iOb2;
        int i;

        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);

        // The following automatically unboxes iOb, performs the increment, and then reboxes
        // the result back into iOb.
        ++iOb;
        System.out.println("After ++iOb: " + iOb);

        // Here, iOb is unboxed, the expression is evaluated, and the result is reboxed and
        // assigned to iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);

        // The same expression is evaluated, but the result is not reboxed.
        i = iOb + (iOb / 3);
        System.out.println("i after expression: " + i);
    }
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
```

In the program, pay special attention to this line:

```java
++iOb;
```

This causes the value in `iOb` to be incremented. It works like this: `iOb` is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```java
class AutoBox4 {
    public static void main(String args[]) {
        Integer iOb = 100;
        Double dOb = 98.6;

        dOb = dOb + iOb;
        System.out.println("dOb after expression: " + dOb);
    }
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the `Double` object `dOb` and the `Integer` object `iOb` participated in the addition, and the result was reboxed and stored in `dOb`.

Because of auto-unboxing, you can use `Integer` numeric objects to control a switch statement. For
example, consider this fragment:

```java
Integer iOb = 2;
switch(iOb) {
    case 1: System.out.println("one");
            break;
    case 2: System.out.println("two");
            break;
    default: System.out.println("error");
}
```

When the switch expression is evaluated, `iOb` is unboxed and its int value is obtained.

As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the past, such code would have involved casts and calls to methods such as `intValue()`.

### Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for `boolean` and `char`. These are `Boolean` and `Character`. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```java
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
    public static void main(String args[]) {

        // Autobox/unbox a boolean.
        Boolean b = true;

        // Below, b is auto-unboxed when used in
        // a conditional expression, such as an if.
        if(b) System.out.println("b is true");
    }
}
```
// Autobox/unbox a char.
Character ch = 'x'; // box a char
char ch2 = ch; // unbox a char

System.out.println("ch2 is " + ch2);
}
}

The output is shown here:

b is true
ch2 is x

The most important thing to notice about this program is the auto-unboxing of b inside the if conditional expression. As you should recall, the conditional expression that controls an if must evaluate to type boolean. Because of auto-unboxing, the boolean value contained within b is automatically unboxed when the conditional expression is evaluated. Thus, with the advent of autoboxing/unboxing, a Boolean object can be used to control an if statement.

Because of auto-unboxing, a Boolean object can now also be used to control any of Java's loop statements. When a Boolean is used as the conditional expression of a while, for, or do/while, it is automatically unboxed into its boolean equivalent. For example, this is now perfectly valid code:

    Boolean b;
    // …
    while(b) { // …

Autoboxing/Unboxing Helps Prevent Errors
In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```java
// An error produced by manual unboxing.
class UnboxingError {
    public static void main(String args[]) {
        Integer iOb = 1000; // autobox the value 1000
        int i = iOb.byteValue(); // manually unbox as byte !!!
        System.out.println(i); // does not display 1000 !
    }
}
```

This program displays not the expected value of 1000, but –24! The reason is that the value inside `iOb` is manually unboxed by calling `byteValue()`, which causes the truncation of the value stored in `iOb`, which is 1,000. This results in the garbage value of –24 being assigned to `i`. Auto-unboxing prevents this type of error because the value in `iOb` will always auto-unbox into a value compatible with `int`.

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, new code
should employ autoboxing/unboxing. It is the way that modern Java code is written.

A Word of Warning

Now that Java includes autoboxing and auto-unboxing, some might be tempted to use objects such as Integer or Double exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing it is possible to write code like this:

```java
// A bad use of autoboxing/unboxing!
Double a, b, c;

a = 10.0;
b = 4.0;

       c = Math.sqrt(a*a + b*b);

System.out.println("Hypotenuse is " + c);
```

In this example, objects of type Double hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type double. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required. Autoboxing/unboxing was not added to Java as a “back
Annotations (Metadata)
Since JDK 5, Java has supported a feature that enables you to embed supplemental information into a source file. This information, called an annotation, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged.

However, this information can be used by various tools during both development and deployment. For example, an annotation might be processed by a source-code generator. The term metadata is also used to refer to this feature, but the term annotation is the most descriptive and more commonly used.

Annotation Basics
An annotation is created through a mechanism based on the interface. Let’s begin with an example. Here is the declaration for an annotation called MyAnno:

```java
// A simple annotation type.
@interface MyAnno {
    String str();
    int val();
}
```

First, notice the `@` that precedes the keyword `interface`. This tells the compiler that an annotation type is being declared. Next, notice the two members `str()` and `val()`. All annotations consist solely of method declarations. However, you don’t provide bodies for
these methods. Instead, Java implements these methods. Moreover, the methods act much like fields, as you will see.

An annotation cannot include an extends clause. However, all annotation types automatically extend the Annotation interface. Thus, Annotation is a superinterface of all annotations. It is declared within the java.lang.annotation package. It overrides hashCode(), equals(), and toString(), which are defined by Object. It also specifies annotationType(), which returns a Class object that represents the invoking annotation.

Once you have declared an annotation, you can use it to annotate a declaration. Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and enum constants can be annotated. Even an annotation can be annotated. In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, you give values to its members. For example, here is an example of MyAnno being applied to a method declaration:

```java
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // …
```

This annotation is linked with the method myMeth(). Look closely at the annotation syntax. The name of the annotation, preceded by an @, is followed by a
parenthesized list of member initializations. To give a member a value, that member’s name is assigned a value. Therefore, in the example, the string “Annotation Example” is assigned to the str member of MyAnno. Notice that no parentheses follow str in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss annotation retention policies. A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the java.lang.annotation.RetentionPolicy enumeration. They are SOURCE, CLASS, and RUNTIME.

An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of CLASS is stored in the .class file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of RUNTIME is stored in the .class file during compilation and is available through the JVM during run time. Thus, RUNTIME retention offers the greatest annotation persistence.
NOTE

An annotation on a local variable declaration is not retained in the .class file.

A retention policy is specified for an annotation by using one of Java’s built-in annotations: @Retention. Its general form is shown here:

```java
@Retention(retention-policy)
@interface MyAnno {
    String str();
    int val();
}
```

Here, retention-policy must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of CLASS is used.

The following version of MyAnno uses @Retention to specify the RUNTIME retention policy. Thus, MyAnno will be available to the JVM during program execution.

```java
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

Obtaining Annotations at Run Time by Use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of RUNTIME, then they can be queried at run time by any Java program through the use of reflection. Reflection is the feature that enables information about a class to be obtained at run time. The
reflection API is contained in the java.lang.reflect package. There are a number of ways to use reflection, and we won’t examine them all here. We will, however, walk through a few examples that apply to annotations.

The first step to using reflection is to obtain a Class object that represents the class whose annotations you want to obtain. Class is one of Java’s built-in classes and is defined in java.lang. It is described in detail in Part II. There are various ways to obtain a Class object. One of the easiest is to call getClass( ), which is a method defined by Object. Its general form is shown here:

```
final Class<?> getClass()
```

It returns the Class object that represents the invoking object.

NOTE

Notice the <?> that follows Class in the declaration of getClass( ) just shown. This is related to Java’s generics feature. getClass( ) and several other reflection-related methods discussed in this chapter make use of generics. Generics are described in Chapter 14. However, an understanding of generics is not needed to grasp the fundamental principles of reflection.

After you have obtained a Class object, you can use its methods to obtain information about the various items
declared by the class, including its annotations. If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item. For example, Class supplies (among others) the `getMethod()`, `getField()`, and `getConstructor()` methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type `Method`, `Field`, and `Constructor`.

To understand the process, let’s work through an example that obtains the annotations associated with a method. To do this, you first obtain a `Class` object that represents the class, and then call `getMethod()` on that `Class` object, specifying the name of the method. `getMethod()` has this general form:

```java
Method getMethod(String methName, Class<?> ... paramTypes)
```

The name of the method is passed in `methName`. If the method has arguments, then `Class` objects representing those types must also be specified by `paramTypes`. Notice that `paramTypes` is a varargs parameter. This means that you can specify as many parameter types as needed, including zero. `getMethod()` returns a `Method` object that represents the method. If the method can’t be found, `NoSuchMethodException` is thrown.

From a `Class`, `Method`, `Field`, or `Constructor` object,
you can obtain a specific annotation associated with that object by calling getAnnotation(). Its general form is shown here:

```java
<A extends Annotation> getAnnotation(Class<A> annoType)
```

Here, annoType is a Class object that represents the annotation in which you are interested. The method returns a reference to the annotation. Using this reference, you can obtain the values associated with the annotation’s members. The method returns null if the annotation is not found, which will be the case if the annotation does not have RUNTIME retention.

Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method:
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // Annotate a method.
    @MyAnno(str = "Annotation Example", val = 100)
    public static void myMeth() {
        Meta ob = new Meta();

        // Obtain the annotation for this method
        // and display the values of the members.
        try {
            // First, get a Class object that represents
            // this class.
            Class<?> c = ob.getClass();

        }
The output from the program is shown here:

Annotation Example 100

This program uses reflection as described to obtain and display the values of str and val in the MyAnno annotation associated with myMeth() in the Meta class. There are two things to pay special attention to. First, in this line

MyAnno anno = m.getAnnotation(MyAnno.class);

notice the expression MyAnno.class. This expression evaluates to a Class object of type MyAnno, the annotation. This construct is called a class literal. You can use this type of expression whenever a Class object of a known class is needed. For example, this statement
could have been used to obtain the Class object for Meta:

```java
Class<?> c = Meta.class;
```

Of course, this approach only works when you know the class name of an object in advance, which might not always be the case. In general, you can obtain a class literal for classes, interfaces, primitive types, and arrays. (Remember, the `<?>` syntax relates to Java's generics feature. It is described in Chapter 14.)

The second point of interest is the way the values associated with `str` and `val` are obtained when they are output by the following line:

```java
System.out.println(anno.str() + " " + anno.val());
```

Notice that they are invoked using the method-call syntax. This same approach is used whenever the value of an annotation member is required.

**A Second Reflection Example**

In the preceding example, `myMeth()` has no parameters. Thus, when `getMethod()` was called, only the name `myMeth` was passed. However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to `getMethod()`. For example, here is a slightly different version of the preceding program:

```java
import java.lang.annotation.*;
import java.lang.reflect.*;
```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {
    // myMeth now has two arguments.
    @MyAnno(str = "Two Parameters", val = 19)
    public static void myMeth(String str, int i) {
        Meta ob = new Meta();
        try {
            Class<?> c = ob.getClass();
            // Here, the parameter types are specified.
            Method m = c.getMethod("myMeth", String.class, int.class);
            MyAnno anno = m.getAnnotation(MyAnno.class);
            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String args[]) {
        myMeth("test", 10);
    }
}

The output from this version is shown here:

Two Parameters 19

In this version, myMeth( ) takes a String and an int parameter. To obtain information about this method, getMethod( ) must be called as shown here:

Method m = c.getMethod("myMeth", String.class, int.class);
Here, the Class objects representing String and int are passed as additional arguments.

Obtaining All Annotations
You can obtain all annotations that have RUNTIME retention that are associated with an item by calling getAnnotations() on that item. It has this general form:

    Annotation[] getAnnotations()

It returns an array of the annotations. getAnnotations() can be called on objects of type Class, Method, Constructor, and Field.

Here is another reflection example that shows how to obtain all annotations associated with a class and with a method. It declares two annotations. It then uses those annotations to annotate a class and a method.
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
    String description();
}

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

    @What(description = "An annotation test method")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() {
        Meta2 ob = new Meta2();

        try {
            Annotation annos[] = ob.getClass().getAnnotations();
        } catch (Exception e) {

        }
    }
}
// Display all annotations for Meta2.
System.out.println("All annotations for Meta2:");
for(Annotation a : annos)
    System.out.println(a);

System.out.println();

// Display all annotations for myMeth.
Method m = ob.getClass().getMethod("myMeth");
annos = m.getAnnotations();

System.out.println("All annotations for myMeth:");
for(Annotation a : annos)
    System.out.println(a);

} catch (NoSuchMethodException exc) {
    System.out.println("Method Not Found.");
}

public static void main(String args[]) {
    myMeth();
}

The output is shown here:

All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)

All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)

The program uses getAnnotations( ) to obtain an array of all annotations associated with the Meta2 class and with the myMeth( ) method. As explained, getAnnotations( ) returns an array of Annotation objects.
Recall that Annotation is a super-interface of all annotation interfaces and that it overrides toString() in Object. Thus, when a reference to an Annotation is output, its toString() method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods getAnnotation() and getAnnotations() used by the preceding examples are defined by the AnnotatedElement interface, which is defined in java.lang.reflect. This interface supports reflection for annotations and is implemented by the classes Method, Field, Constructor, Class, and Package.

In addition to getAnnotation() and getAnnotations(), AnnotatedElement defines two other methods. The first is getDeclaredAnnotations(), which has this general form:

    Annotation[] getDeclaredAnnotations()

It returns all non-inherited annotations present in the invoking object. The second is isAnnotationPresent(), which has this general form:

    boolean isAnnotationPresent(Class<? extends Annotation> annoType)

It returns true if the annotation specified by annoType is
associated with the invoking object. It returns false otherwise.

Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a default clause to a member’s declaration. It has this general form:

```plaintext
type member( ) default value;
```

Here, value must be of a type compatible with type.

Here is `@MyAnno` rewritten to include default values:

```java
// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testing";
  int val() default 9000;
}
```

This declaration gives a default value of “Testing” to `str` and 9000 to `val`. This means that neither value needs to be specified when `@MyAnno` is used. However, either or both can be given values if desired. Therefore, following are the four ways that `@MyAnno` can be used:

```java
@MyAnno() // both str and val default
@MyAnno(str = "some string") // val defaults
@MyAnno(val = 100) // str defaults
@MyAnno(str = "Testing", val = 100) // no defaults
```

The following program demonstrates the use of
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

class Meta3 {

    // Annotate a method using the default values.
    @MyAnno()
    public static void myMeth() {
        Meta3 ob = new Meta3();

        // Obtain the annotation for this method
        // and display the values of the members.
        try {
            Class<?> c = ob.getClass();

            Method m = c.getMethod("myMeth");

            MyAnno anno = m.getAnnotation(MyAnno.class);

            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String args[]) {
        myMeth();
    }
}

The output is shown here:

Testing 9000
Marker Annotations

A marker annotation is a special kind of annotation that contains no members. Its sole purpose is to mark a declaration. Thus, its presence as an annotation is sufficient. The best way to determine if a marker annotation is present is to use the method `isAnnotationPresent( )`, which is defined by the `AnnotatedElement` interface.

Here is an example that uses a marker annotation. Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker {}

class Marker {

    // Annotate a method using a marker.
    // Notice that no ( ) is needed.
    @MyMarker
    public static void myMeth() {
        Marker ob = new Marker();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            // Determine if the annotation is present.
            if(m.isAnnotationPresent(MyMarker.class))
                System.out.println("MyMarker is present.");
        }
    }
}
```
The output, shown here, confirms that `@MyMarker` is present:

MyMarker is present.

In the program, notice that you do not need to follow `@MyMarker` with parentheses when it is applied. Thus, `@MyMarker` is applied simply by using its name, like this:

`@MyMarker`

It is not wrong to supply an empty set of parentheses, but they are not needed.

Single-Member Annotations

A single-member annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don’t need to specify the name of the member. However, in order to use this shorthand, the name of the member must be `value`.

Here is an example that creates and uses a single-member annotation:
As expected, this program displays the value 100. In the program, `@MySingle` is used to annotate `myMeth()`, as shown here:

```java
@MySingle(100)
public static void myMeth() {...
```

Notice that `value` need not be specified.

You can use the single-value syntax when applying an
annotation that has other members, but those other members must all have default values. For example, here the value `xyz` is added, with a default value of zero:

```java
@interface SomeAnno {
    int value();
    int xyz() default 0;
}
```

In cases in which you want to use the default for `xyz`, you can apply `@SomeAnno`, as shown next, by simply specifying the value of `value` by using the single-member syntax.

```java
@SomeAnno(88)
```

In this case, `xyz` defaults to zero, and `value` gets the value 88. Of course, to specify a different value for `xyz` requires that both members be explicitly named, as shown here:

```java
@SomeAnno(value = 88, xyz = 99)
```

Remember, whenever you are using a single-member annotation, the name of that member must be `value`.

The Built-In Annotations

Java defines many built-in annotations. Most are specialized, but eight are general purpose. Of these, four are imported from `java.lang.annotation`: `@Retention`, `@Documented`, `@Target`, and `@Inherited`. Four—`@Override`, `@Deprecated`, `@SafeVarargs`, and `@SuppressWarnings`—are included in `java.lang`. Each is described here.
@Retention
@Retention is designed to be used only as an annotation to another annotation. It specifies the retention policy as described earlier in this chapter.

@Documented
The @Documented annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.

@Target
The @Target annotation specifies the types of declarations to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. @Target takes one argument, which must be a constant from the ElementType enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond:

<table>
<thead>
<tr>
<th>Target Constant</th>
<th>Annotation Can Be Applied To</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANNOTATION_TYPE</td>
<td>Another annotation</td>
</tr>
<tr>
<td>CONSTRUCTOR</td>
<td>Constructor</td>
</tr>
<tr>
<td>FIELD</td>
<td>Field</td>
</tr>
<tr>
<td>LOCAL_VARIABLE</td>
<td>Local variable</td>
</tr>
<tr>
<td>METHOD</td>
<td>Method</td>
</tr>
<tr>
<td>PACKAGE</td>
<td>Package</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>Parameter</td>
</tr>
<tr>
<td>TYPE</td>
<td>Class, interface, or enumeration</td>
</tr>
</tbody>
</table>
You can specify one or more of these values in a @Target annotation. To specify multiple values, you must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this @Target annotation:

```java
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

@Inherited

@Inherited is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. @Inherited causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with @Inherited, then that annotation will be returned.

@Override

@Override is a marker annotation that can be used only on methods. A method annotated with @Override must override a method from a superclass. If it doesn’t, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.
@Deprecated
@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.

@SafeVarargs
@SafeVarargs is a marker annotation that can be applied to methods and constructors. It indicates that no unsafe actions related to a varargs parameter occur. It is used to suppress unchecked warnings on otherwise safe code as it relates to non-reifiable vararg types and parameterized array instantiation. (A non-reifiable type is, essentially, a generic type. Generics are described in Chapter 14.) It must be applied only to vararg methods or constructors that are static or final. It was added by JDK 7.

@SuppressWarnings
@SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form. This annotation can be applied to any type of declaration.

Some Restrictions
There are a number of restrictions that apply to annotation declarations. First, no annotation can inherit another. Second, all methods declared by an annotation must be without parameters. Furthermore, they must
return one of the following:

- A primitive type, such as int or double
- An object of type String or Class
- An enum type
- Another annotation type
- An array of one of the preceding types

Annotations cannot be generic. In other words, they cannot take type parameters. (Generics are described in Chapter 14.) Finally, annotation methods cannot specify a throws clause.
This chapter introduces two of Java’s most important packages: io and applet. The io package supports Java’s basic I/O (input/output) system, including file I/O. The applet package supports applets. Support for both I/O and applets comes from Java’s core API libraries, not from language keywords. For this reason, an in-depth discussion of these topics is found in Part II of this book, which examines Java’s API classes. This chapter discusses the foundation of these two subsystems so that you can see how they are integrated into the Java language and how they fit into the larger context of the Java programming and execution environment. This chapter also examines JDK 7’s new try-with-resources statement and the last of Java’s keywords: transient, volatile, instanceof, native, strictfp, and assert. It concludes by examining static import and by describing another use for the this keyword.

I/O Basics
As you may have noticed while reading the preceding 12 chapters, not much use has been made of I/O in the example programs. In fact, aside from print( ) and println( ), none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are either graphically oriented programs that rely on Java’s Abstract Window Toolkit (AWT) or Swing for user interaction, or they are Web applications. Although
text-based, console programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java’s support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not that useful in real-world Java programming.

The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java’s I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master. A general overview of I/O is presented here. A detailed description is found in Chapters 19 and 20.

Streams
Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io
NOTE

In addition to the stream-based I/O defined in java.io, Java also provides buffer- and channel-based I/O, which is defined in java.nio and its subpackages. They are described in Chapter 20.

Byte Streams and Character Streams

Java defines two types of streams: byte and character. Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. Although old code that doesn’t use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.
An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes
Byte streams are defined by using two class hierarchies. At the top are two abstract classes: InputStream and OutputStream. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The byte stream classes in java.io are shown in Table 13-1. A few of these classes are discussed later in this section. Others are described in Part II of this book. Remember, to use the stream classes, you must import java.io.
The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement. Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data. Each has forms that are abstract and must be overridden by derived stream classes.

### The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: Reader and Writer. These abstract classes handle Unicode...
Character streams. Java has several concrete subclasses of each of these. The character stream classes in java.io are shown in Table 13-2.

<table>
<thead>
<tr>
<th>Stream Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferedReader</td>
<td>Buffered input character stream</td>
</tr>
<tr>
<td>BufferedWriter</td>
<td>Buffered output character stream</td>
</tr>
<tr>
<td>CharArrayReader</td>
<td>Input stream that reads from a character array</td>
</tr>
<tr>
<td>CharArrayWriter</td>
<td>Output stream that writes to a character array</td>
</tr>
<tr>
<td>FileReader</td>
<td>Input stream that reads from a file</td>
</tr>
<tr>
<td>FileWriter</td>
<td>Output stream that writes to a file</td>
</tr>
<tr>
<td>FilterReader</td>
<td>Filtered reader</td>
</tr>
<tr>
<td>FilterWriter</td>
<td>Filtered writer</td>
</tr>
<tr>
<td>InputStreamReader</td>
<td>Input stream that translates bytes to characters</td>
</tr>
<tr>
<td>LineNumberReader</td>
<td>Input stream that counts lines</td>
</tr>
<tr>
<td>OutputStreamWriter</td>
<td>Output stream that translates characters to bytes</td>
</tr>
<tr>
<td>PipedReader</td>
<td>Input pipe</td>
</tr>
<tr>
<td>PipedWriter</td>
<td>Output pipe</td>
</tr>
<tr>
<td>PrintWriter</td>
<td>Output stream that contains print() and println()</td>
</tr>
<tr>
<td>PushbackReader</td>
<td>Input stream that allows characters to be returned to the input stream</td>
</tr>
<tr>
<td>Reader</td>
<td>Abstract class that describes character stream input</td>
</tr>
<tr>
<td>StringReader</td>
<td>Input stream that reads from a string</td>
</tr>
<tr>
<td>StringWriter</td>
<td>Output stream that writes to a string</td>
</tr>
<tr>
<td>Writer</td>
<td>Abstract class that describes character stream output</td>
</tr>
</tbody>
</table>

Table 13-2 The Character Stream I/O Classes in java.io

The abstract classes Reader and Writer define several key methods that the other stream classes implement. Two of the most important methods are read() and write(), which read and write characters of data, respectively. Each has forms that are abstract and must be overridden by derived stream classes.

The Predefined Streams

As you know, all Java programs automatically import
the java.lang package. This package defines a class called System, which encapsulates several aspects of the runtime environment. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. System also contains three predefined stream variables: in, out, and err. These fields are declared as public, static, and final within System. This means that they can be used by any other part of your program and without reference to a specific System object.

System.out refers to the standard output stream. By default, this is the console. System.in refers to standard input, which is the keyboard by default. System.err refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream. These are byte streams, even though they are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

The preceding chapters have been using System.out in their examples. You can use System.err in much the same way. As explained in the next section, use of System.in is a little more complicated.

Reading Console Input
In Java 1.0, the only way to perform console input was to use a byte stream. Today, using a byte stream to read
console input is still acceptable. However, for commercial applications, the preferred method of reading console input is to use a character-oriented stream. This makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from System.in. To obtain a character-based stream that is attached to the console, wrap System.in in a BufferedReader object. BufferedReader supports a buffered input stream. A commonly used constructor is shown here:

```
BufferedReader(Reader inputReader)
```

Here, inputReader is the stream that is linked to the instance of BufferedReader that is being created. Reader is an abstract class. One of its concrete subclasses is InputStreamReader, which converts bytes to characters. To obtain an InputStreamReader object that is linked to System.in, use the following constructor:

```
InputStreamReader(InputStream inputStream)
```

Because System.in refers to an object of type InputStream, it can be used for inputStream. Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

After this statement executes, br is a character-based stream that is linked to the console through System.in.
Reading Characters

To read a character from a BufferedReader, use `read()`.

The version of `read()` that we will be using is

```java
int read() throws IOException
```

Each time that `read()` is called, it reads a character from the input stream and returns it as an integer value. It returns -1 when the end of the stream is encountered. As you can see, it can throw an `IOException`.

The following program demonstrates `read()` by reading characters from the console until the user types a “q.” Notice that any I/O exceptions that might be generated are simply thrown out of `main()`.

Such an approach is common when reading from the console in simple example programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```java
// Use a BufferedReader to read characters from the console.
import java.io.*;

class BRRead {
    public static void main(String args[]) throws IOException {
        char c;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        // read characters
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

Here is a sample run:
This output may look a little different from what you expected because System.in is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make read( ) particularly valuable for interactive console input.

Reading Strings
To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

    String readLine( ) throws IOException

As you can see, it returns a String object.

The following program demonstrates BufferedReader and the readLine( ) method; the program reads and displays lines of text until you enter the word “stop”:

    // Read a string from console using a BufferedReader.  
    import java.io.*;  
    class BRReadLines {  
        public static void main(String args[]) throws IOException  
        {  
            // create a BufferedReader using System.in  
            BufferedReader br = new BufferedReader(new 
                 InputStreamReader(System.in));  
            String str;  
            System.out.println("Enter lines of text.");
    
    
}
System.out.println("Enter 'stop' to quit.");
do {
    str = br.readLine();
    System.out.println(str);
} while(!str.equals("stop"));

The next example creates a tiny text editor. It creates an array of String objects and then reads in lines of text, storing each line in the array. It will read up to 100 lines or until you enter "stop." It uses a BufferedReader to read from the console.

// A tiny editor.
import java.io.*;

class TinyEdit {
    public static void main(String args[]) throws IOException {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str[] = new String[100];
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        for(int i=0; i<100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("stop")) break;
        }
        System.out.println("\nHere is your file:");
        // display the lines
        for(int i=0; i<100; i++) {
            if(str[i].equals("stop")) break;
            System.out.println(str[i]);
        }
    }
}

Here is a sample run:

Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Writing Console Output

Console output is most easily accomplished with print( ) and println( ), described earlier, which are used in most of the examples in this book. These methods are defined by the class PrintStream (which is the type of object referenced by System.out). Even though System.out is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is shown here:

```java
void write(int byteval)
```

This method writes the byte specified by byteval. Although byteval is declared as an integer, only the low-order eight bits are written. Here is a short example that uses write( ) to output the character “A” followed by a newline to the screen:

```java
// Demonstrate System.out.write().
class WriteDemo {
    public static void main(String args[]) {
        int b;

        b = 'A';
```
System.out.write(b);
System.out.write('n');
}
}

You will not often use `write( )` to perform console output (although doing so might be useful in some situations) because `print( )` and `println( )` are substantially easier to use.

The PrintWriter Class

Although using `System.out` to write to the console is acceptable, its use is probably best for debugging purposes or for sample programs, such as those found in this book. For real-world programs, the recommended method of writing to the console when using Java is through a `PrintWriter` stream. `PrintWriter` is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier.

`PrintWriter` defines several constructors. The one we will use is shown here:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

Here, `outputStream` is an object of type `OutputStream`, and `flushOnNewline` controls whether Java flushes the output stream every time a `println( )` method is called. If `flushOnNewline` is true, flushing automatically takes place. If false, flushing is not automatic.

`PrintWriter` supports the `print( )` and `println( )` methods. Thus, you can use these methods in the same
way as you used them with System.out. If an argument is not a simple type, the PrintWriter methods call the object’s toString( ) method and then print the result.

To write to the console by using a PrintWriter, specify System.out for the output stream and flush the stream after each newline. For example, this line of code creates a PrintWriter that is connected to console output:

    PrintWriter pw = new PrintWriter(System.out, true);

The following application illustrates using a PrintWriter to handle console output:

    // Demonstrate PrintWriter
    import java.io.*;

    public class PrintWriterDemo {
        public static void main(String args[]) {
            PrintWriter pw = new PrintWriter(System.out, true);

            pw.println("This is a string");
            int i = -7;
            pw.println(i);
            double d = 4.5e-7;
            pw.println(d);
        }
    }

The output from this program is shown here:

    This is a string
    -7
    4.5E-7

Remember, there is nothing wrong with using System.out to write simple text output to the console when you are learning Java or debugging your programs. However, using a PrintWriter makes your real-world applications easier to internationalize. Because no
advantage is gained by using a PrintWriter in the sample programs shown in this book, we will continue to use System.out to write to the console.

Reading and Writing Files
Java provides a number of classes and methods that allow you to read and write files. Before we begin, it is important to state that the topic of file I/O is quite large and file I/O is examined in detail in Part II. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although bytes streams are used, these techniques can be adapted to the character-based streams.

Two of the most often-used stream classes are FileInputStream and FileOutputStream, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:

```java
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

Here, fileName specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then FileNotFoundException is thrown. For output streams, if the file cannot be opened or created, then FileNotFoundException is thrown.
FileNotFoundException is a subclass of IOException. When an output file is opened, any preexisting file by the same name is destroyed.

NOTE
In situations in which a security manager is present, several of the file classes, including FileInputStream and FileOutputStream, will throw a SecurityException if a security violation occurs when attempting to open a file. By default, applications run via java do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible SecurityException. However, other types of applications (such as applets) will use the security manager, and file I/O performed by such an application could generate a SecurityException. In that case, you will need to appropriately handle this exception.

When you are done with a file, you must close it. This is done by calling the close() method, which is implemented by both FileInputStream and FileOutputStream. It is shown here:

```java
void close( ) throws IOException
```
Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in “memory leaks” because of unused resources remaining allocated.

NOTE
Beginning with JDK 7, the close() method is specified by the AutoCloseable interface in java.lang. AutoCloseable is inherited by the Closeable interface in java.io. Both interfaces are implemented by the stream classes, including FileInputStream and FileOutputStream.

Before moving on, it is important to point out that there are two basic approaches that you can use to close a file when you are done with it. The first is the traditional approach, in which close() is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7 and is, therefore, found in all legacy code. The second is to use the new try-with-resources statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to close() is executed. Since there are millions of lines of pre-JDK 7 legacy code that are still being used and maintained, it is important that you know and understand the traditional approach. Therefore, we will begin with it. The new automated approach is described in the following section.

To read from a file, you can use a version of read() that is defined within FileInputStream. The one that we will use is shown here:

```java
int read() throws IOException
```

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value.
returns -1 when the end of the file is encountered. It can throw an IOException.

The following program uses read( ) to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

/* Display a text file.
   To use this program, specify the name of the file that you want to see.
   For example, to see a file called TEST.TXT, use the following command line.

   java ShowFile TEST.TXT
*/

import java.io.*;

class ShowFile {
    public static void main(String args[]) {
        int i;
        FileInputStream fin;

        // First, confirm that a filename has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // Attempt to open the file.
        try {
            fin = new FileInputStream(args[0]);
        } catch(FileNotFoundException e) {
            System.out.println("Cannot Open File");
            return;
        }
    }
}
In the program, notice the try/catch blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled. Be aware that in simple programs or example code, it is common to see I/O exceptions simply thrown out of main( ), as was done in the earlier console I/O examples. Also, in some real-world code, it can be helpful to let an exception propagate to a calling routine to let the caller know that an I/O operation failed. However, most of the file I/O examples in this book handle all I/O exceptions explicitly, as shown, for the sake of illustration.

Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call close( ) within a finally block. In this approach, all of the methods that access the file are contained within a try block, and the finally
block is used to close the file. This way, no matter how the try block terminates, the file is closed. Assuming the preceding example, here is how the try block that reads the file can be recoded:

```java
try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);
} catch(IOException e) {
    System.out.println("Error Reading File");
} finally {
    // Close file on the way out of the try block.
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}
```

Although not an issue in this case, one advantage to this approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the finally block.

Sometimes it’s easier to wrap the portions of a program that open the file and access the file within a single try block (rather than separating the two) and then use a finally block to close the file. For example, here is another way to write the ShowFile program:
/* Display a text file. 
   To use this program, specify the name 
of the file that you want to see. 
For example, to see a file called TEST.TXT, 
use the following command line. 

java ShowFile TEST.TXT 

This variation wraps the code that opens and 
accesses the file within a single try block. 
The file is closed by the finally block. 
*/

import java.io.*;
class ShowFile {
    public static void main(String args[]) 
    {
        int i;
        FileInputStream fin = null;

        // First, confirm that a filename has been specified.
        if (args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }
In this approach, notice that fin is initialized to null. Then, in the finally block, the file is closed only if fin is not null. This works because fin will be non-null only if the file is successfully opened. Thus, close() is not called if an exception occurs while opening the file.

It is possible to make the try/catch sequence in the preceding example a bit more compact. Because FileNotFoundException is a subclass of IOException, it need not be caught separately. For example, here is the sequence recoded to eliminate catching FileNotFoundException. In this case, the standard exception message, which describes the error, is displayed.

```java
try {
    fin = new FileInputStream(args[0]);

    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch (FileNotFoundException e) {
    System.out.println("File Not Found.");
} catch (IOException e) {
    System.out.println("An I/O Error Occurred");
} finally {
    // Close file in all cases.
    try {
        if(fin != null) fin.close();
    } catch (IOException e) {
        System.out.println("Error Closing File");
    }
}
```
In this approach, any error, including an error opening the file, is simply handled by the single catch statement. Because of its compactness, this approach is used by many of the I/O examples in this book. Be aware, however, that this approach is not appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused if a user mistypes a filename. In such a situation, you might want to prompt for the correct name, for example, before entering a try block that accesses the file.

To write to a file, you can use the write( ) method defined by FileOutputStream. Its simplest form is shown here:

```java
void write(int byteval) throws IOException
```

This method writes the byte specified by byteval to the file. Although byteval is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an IOException is thrown. The
next example uses write( ) to copy a file:

```java
/* Copy a file.
 To use this program, specify the name
 of the source file and the destination file.
 For example, to copy a file called FIRST.TXT
to a file called SECOND.TXT, use the following
command line.

 java CopyFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // First, confirm that both files have been specified.
        if(args.length != 2) {
            System.out.println("Usage: CopyFile from to");
            return;
        }
```
In the program, notice that two separate try blocks are used when closing the files. This ensures that both files are closed, even if the call to `fin.close()` throws an exception.

In general, notice that all potential I/O errors are handled in the preceding two programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. Not only do exceptions make file handling cleaner, but they also enable Java to easily differentiate the end-of-file condition from file errors when input is being
performed. In C/C++, many input functions return the same value when an error occurs and when the end of the file is reached. (That is, in C/C++, an EOF condition often is mapped to the same value as an input error.) This usually means that the programmer must include extra program statements to determine which event actually occurred. In Java, input errors are passed to your program via exceptions, not by values returned by read(). Thus, when read() returns -1, it means only one thing: the end of the file has been encountered.

Automatically Closing a File
In the preceding section, the example programs have made explicit calls to close() to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7. Although this approach is still valid and useful, JDK 7 adds a new feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as automatic resource management, or ARM for short, is based on an expanded version of the try statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks, and could lead to other problems.

Automatic resource management is based on an expanded form of the try statement. Here is its general form:
try (resource-specification) {
  // use the resource
}

Here, resource-specification is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the try block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call close() explicitly.) Of course, this form of try can also include catch and finally clauses. This new form of try is called the try-with-resources statement.

The try-with-resources statement can be used only with those resources that implement the AutoCloseable interface defined by java.lang. This interface defines the close() method. AutoCloseable is inherited by the Closeable interface in java.io. Both interfaces are implemented by the stream classes. Thus, try-with-resources can be used when working with streams, including file streams.

As a first example of automatically closing a file, here is a reworked version of the ShowFile program that uses it:

/* This version of the ShowFile program uses a try-with-resources statement to automatically close a file after it is no longer needed.
   Note: This code requires JDK 7 or later.
*/

import java.io.*;
class ShowFile {
    public static void main(String args[]) {
        int i;

        // First, confirm that a filename has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // The following code uses a try-with-resources statement to open
        // a file and then automatically close it when the try block is left.
        try(
            FileInputStream fin = new FileInputStream(args[0])
        ) {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);
        } catch(FileNotFoundException e) {
            System.out.println("File Not Found.");
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}

In the program, pay special attention to how the file is
opened within the try statement:

    try(FileInputStream fin = new FileInputStream(args[0])) {
        //
        do {
            i = fin.read();
            if(i != -1) System.out.print((char) i);
        } while(i != -1);
    }

Notice how the resource-specification portion of the try
declares a FileInputStream called fin, which is then
assigned a reference to the file opened by its constructor.
Thus, in this version of the program, the variable fin is
local to the try block, being created when the try is
entered. When the try is left, the stream associated with
fin is automatically closed by an implicit call to close().
You don’t need to call close() explicitly, which means
that you can’t forget to close the file. This is a key
It is important to understand that the resource declared in the try statement is implicitly final. This means that you can’t assign to the resource after it has been created. Also, the scope of the resource is limited to the try-with-resources statement.

You can manage more than one resource within a single try statement. To do so, simply separate each resource specification with a semicolon. The following program shows an example. It reworks the CopyFile program shown earlier so that it uses a single try-with-resources statement to manage both fin and fout.

```java
/* A version of CopyFile that uses try-with-resources. It demonstrates two resources (in this case files) being managed by a single try statement. */

import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException {
        int i;

        // First, confirm that both files have been specified.
        if(args.length != 2) {
            System.out.println("Usage: CopyFile from to");
            return;
        }

        // Open and manage two files via the try statement.
        try (FileInputStream fin = new FileInputStream(args[0]);
             FileOutputStream fout = new FileOutputStream(args[1])) {
            do {
                i = fin.read();
                if(i != -1) fout.write(i);
            } while(i != -1);
        }
    }
}
```
In this program, notice how the input and output files are opened within the try block:

```java
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
    // …
}
```

After this try block ends, both `fin` and `fout` will have been closed. If you compare this version of the program to the previous version, you will see that it is much shorter. The ability to streamline source code is a side-benefit of automatic resource management.

There is one other aspect to try-with-resources that needs to be mentioned. In general, when a try block executes, it is possible that an exception inside the try block will lead to another exception that occurs when the resource is closed in a finally clause. In the case of a “normal” try statement, the original exception is lost, being preempted by the second exception. However, when using try-with-resources, the second exception is suppressed. It is not, however, lost. Instead, it is added to the list of suppressed exceptions associated with the first exception. The list of suppressed exceptions can be obtained by using the `getSuppressed()` method defined by `Throwable`.

Because of the benefits that the try-with-resources statement offers, it will be used by many, but not all, of
Some of the examples will still use the traditional approach to closing a resource. There are several reasons for this. First, there are millions of lines of legacy code in widespread use that rely on the traditional approach. It is important that all Java programmers be fully versed in, and comfortable with, the traditional approach when maintaining this older code. Second, because not all project development will immediately switch to a new version of the JDK, it is likely that some programmers will continue to work in a pre-JDK 7 environment for a period of time. In such situations, the expanded form of try is not available. Finally, there may be cases in which explicitly closing a resource is more appropriate than the automated approach. For these reasons, some of the examples in this book will continue to use the traditional approach, explicitly calling close(). In addition to illustrating the traditional technique, these examples can also be compiled and run by all readers in all environments.

REMEMBER

A few examples in this book use the traditional approach to closing files as a means of illustrating this technique, which is widely used in legacy code. However, for new code, you will usually want to use the new automated approach supported by the try-with-resources statement just described.

Applet Fundamentals
All of the preceding examples in this book have been
Java console-based applications. However, these types of applications constitute only one class of Java programs. Another type of program is the applet. As mentioned in Chapter 1, applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

Many of the issues connected with the creation and use of applets are found in Part II, when the applet package is examined, and also when Swing is described in Part III. However, the fundamentals connected to the creation of an applet are presented here, because applets are not structured in the same way as the programs that have been used thus far. As you will see, applets differ from console-based applications in several key areas.

Let’s begin with the simple applet shown here:

```java
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

This applet begins with two import statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-
based I/O classes. The AWT contains support for a window-based, graphical user interface. As you might expect, the AWT is quite large and sophisticated, and a complete discussion of it consumes several chapters in Part II of this book. Fortunately, this simple applet makes very limited use of the AWT. (Applets can also use Swing to provide the graphical user interface, but this approach is described later in this book.) The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be a subclass (either directly or indirectly) of Applet.

The next line in the program declares the class SimpleApplet. This class must be declared as public, because it will be accessed by code that is outside the program.

Inside SimpleApplet, paint() is declared. This method is defined by the AWT and must be overridden by the applet. paint() is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. paint() is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint() is called. The paint() method has one parameter of type Graphics. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the
applet is required.

Inside `paint()` is a call to `drawString()`, which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

```java
void drawString(String message, int x, int y)
```

Here, `message` is the string to be output beginning at `x,y`. In a Java window, the upper-left corner is location 0,0. The call to `drawString()` in the applet causes the message “A Simple Applet” to be displayed beginning at location 20,20.

Notice that the applet does not have a `main()` method. Unlike Java programs, applets do not begin execution at `main()`. In fact, most applets don’t even have a `main()` method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

After you enter the source code for `SimpleApplet`, compile in the same way that you have been compiling programs. However, running `SimpleApplet` involves a different process. In fact, there are two ways in which you can run an applet:

- Executing the applet within a Java-compatible web browser.
- Using an applet viewer, such as the standard tool, `appletviewer`. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.
Each of these methods is described next.

One way to execute an applet in a web browser is to write a short HTML text file that contains a tag that loads the applet. Currently, Oracle recommends using the APPLET tag for this purpose. (The OBJECT tag can also be used. See Chapter 22 for further information regarding applet deployment strategies.) Using APPLET, here is the HTML file that executes SimpleApplet:

```html
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The width and height statements specify the dimensions of the display area used by the applet. (The APPLET tag contains several other options that are examined more closely in Part II.) After you create this file, you can execute your browser and then load this file, which causes SimpleApplet to be executed.

To execute SimpleApplet with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called RunApp.html, then the following command line will run SimpleApplet:

```
C:\>appletviewer RunApp.html
```

However, a more convenient method exists that you can use to speed up testing. Simply include a comment at the head of your Java source code file that contains the APPLET tag. By doing so, your code is documented with a prototype of the necessary HTML statements, and you can test your compiled applet merely by starting the
applet viewer with your Java source code file. If you use this method, the SimpleApplet source file looks like this:

```java
import java.awt.*;
import java.applet.*;

/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

With this approach, you can quickly iterate through applet development by using these three steps:

1. Edit a Java source file.
2. Compile your program.
3. Execute the applet viewer, specifying the name of your applet’s source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

The window produced by SimpleApplet, as displayed by the applet viewer, is shown in the following illustration:

While the subject of applets is more fully discussed later in this book, here are the key points that you
should remember now:

• Applets do not need a main( ) method.

• Applets must be run under an applet viewer or a Java-compatible browser.

• User I/O is not accomplished with Java’s stream I/O classes. Instead, applets use the interface provided by the AWT or Swing.

The transient and volatile Modifiers

Java defines two interesting type modifiers: transient and volatile. These modifiers are used to handle somewhat specialized situations.

When an instance variable is declared as transient, then its value need not persist when an object is stored. For example:

```java
class T {
    transient int a; // will not persist
    int b; // will persist
}
```

Here, if an object of type T is written to a persistent storage area, the contents of a would not be saved, but the contents of b would.

The volatile modifier tells the compiler that the variable modified by volatile can be changed unexpectedly by other parts of your program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or master) copy of the variable
is updated at various times, such as when a synchronized method is entered. While this approach works fine, it may be inefficient at times. In some cases, all that really matters is that the master copy of a variable always reflects its current state. To ensure this, simply specify the variable as volatile, which tells the compiler that it must always use the master copy of a volatile variable (or, at least, always keep any private copies up-to-date with the master copy, and vice versa). Also, accesses to the master variable must be executed in the precise order in which they are executed on any private copy.

Using instanceof

Sometimes, knowing the type of an object during run time is useful. For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object’s type at run time is important involves casting. In Java, an invalid cast causes a run-time error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts that can be detected only at run time. For example, a superclass called A can produce two subclasses, called B and C. Thus, casting a B object into type A or casting a C object into type A is legal, but casting a B object into type C (or vice versa) isn’t legal. Because an object of type A can refer to objects of either B or C, how can you know, at run time, what type of object is actually being referred to before
attempting the cast to type C? It could be an object of type A, B, or C. If it is an object of type B, a run-time exception will be thrown. Java provides the run-time operator instanceof to answer this question.

The instanceof operator has this general form:

```java
objref instanceof type
```

Here, `objref` is a reference to an instance of a class, and `type` is a class type. If `objref` is of the specified type or can be cast into the specified type, then the `instanceof` operator evaluates to true. Otherwise, its result is false. Thus, `instanceof` is the means by which your program can obtain run-time type information about an object.

The following program demonstrates `instanceof`:
// Demonstrate instanceof operator.
class A {
    int i, j;
}

class B {
    int i, j;
}

class C extends A {
    int k;
}

class D extends A {
    int k;
}

class InstanceOf {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();
        if(a instanceof A)
            System.out.println("a is instance of A");
        if(b instanceof B)
            System.out.println("b is instance of B");
        if(c instanceof C)
            System.out.println("c is instance of C");
        if(c instanceof A)
            System.out.println("c can be cast to A");
        if(a instanceof C)
            System.out.println("a can be cast to C");
        System.out.println();
    }
}
The output from this program is shown here:

a is instance of A
b is instance of B
c is instance of C
c can be cast to A

ob now refers to d
ob is instance of D

ob now refers to c
ob cannot be cast to D
ob can be cast to A
The instanceof operator isn’t needed by most programs, because, generally, you know the type of object with which you are working. However, it can be very useful when you’re writing generalized routines that operate on objects of a complex class hierarchy.

strictfp
A relatively new keyword is strictfp. With the creation of Java 2, the floating-point computation model was relaxed slightly. Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation. This prevents overflow or underflow in some cases. By modifying a class, a method, or interface with strictfp, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java. When a class is modified by strictfp, all the methods in the class are also modified by strictfp automatically.

For example, the following fragment tells Java to use the original floating-point model for calculations in all methods defined within MyClass:

```java
strictfp class MyClass { //...
```

Frankly, most programmers never need to use strictfp, because it affects only a very small class of problems.

Native Methods
Although it is rare, occasionally you may want to call a
subroutine that is written in a language other than Java. Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code. For example, you may want to call a native code subroutine to achieve faster execution time. Or, you may want to use a specialized, third-party library, such as a statistical package. However, because Java programs are compiled to bytecode, which is then interpreted (or compiled on-the-fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the native keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the native modifier, but do not define any body for the method. For example:

```
public native int meth();
```

After you declare a native method, you must write the native method and follow a rather complex series of steps to link it with your Java code.

Most native methods are written in C. The mechanism used to integrate C code with a Java program is called the Java Native Interface (JNI). A detailed description of the JNI is beyond the scope of this book, but the following description provides sufficient information for most applications.
NOTE

The precise steps that you need to follow will vary between different Java environments. They also depend on the language that you are using to implement the native method. The following discussion assumes a Windows environment. The language used to implement the native method is C.

The easiest way to understand the process is to work through an example. To begin, enter the following short program, which uses a native method called test( ):

```java
// A simple example that uses a native method.
public class NativeDemo {
    int i;
    public static void main(String args[]) {
        NativeDemo ob = new NativeDemo();

        ob.i = 10;
        System.out.println("This is ob.i before the native method:" + ob.i);
        ob.test(); // call a native method
        System.out.println("This is ob.i after the native method:" + ob.i);
    }

    // declare native method
    public native void test();

    // load DLL that contains static method
    static {
        System.loadLibrary("NativeDemo");
    }
}
```

Notice that the test( ) method is declared as native and has no body. This is the method that we will implement in C shortly. Also notice the static block. As explained earlier in this book, a static block is executed only once,
when your program begins execution (or, more precisely, when its class is first loaded). In this case, it is used to load the dynamic link library that contains the native implementation of test( ). (You will see how to create this library soon.)

The library is loaded by the loadLibrary( ) method, which is part of the System class. This is its general form:

```java
static void loadLibrary(String filename)
```

Here, filename is a string that specifies the name of the file that holds the library. For the Windows environment, this file is assumed to have the .DLL extension.

After you enter the program, compile it to produce NativeDemo.class. Next, you must use javah.exe to produce one file: NativeDemo.h. (javah.exe is included in the JDK.) You will include NativeDemo.h in your implementation of test( ). To produce NativeDemo.h, use the following command:

```
javah -jni NativeDemo
```

This command produces a header file called NativeDemo.h. This file must be included in the C file that implements test( ). The output produced by this command is shown here:

```c
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifdef __cplusplus
extern "C" {
#endif
```
Pay special attention to the following line, which defines the prototype for the test() function that you will create:

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

Notice that the name of the function is `Java_NativeDemo_test()`. You must use this as the name of the native function that you implement. That is, instead of creating a C function called `test()`, you will create one called `Java_NativeDemo_test()`. The `NativeDemo` component of the prefix is added because it identifies the `test()` method as being part of the `NativeDemo` class. Remember, another class may define its own native `test()` method that is completely different from the one declared by `NativeDemo`. Including the class name in the prefix provides a way to differentiate between differing versions. As a general rule, native functions will be given a name whose prefix includes the name of the class in which they are declared.

After producing the necessary header file, you can write your implementation of `test()` and store it in a file named `NativeDemo.c`: 

```c
#ifndef
/*
 * Class:     NativeDemo
 * Method:    test
 * Signature: ()V
 */

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
#endif
```
/* This file contains the C version of the test() method. */

#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj) {
    jclass cls;
    jfieldID fid;
    jint i;

    printf("Starting the native method.\n");
    cls = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env, cls, "i", "I");

    if(fid == 0) {
        printf("Could not get field id.\n");
        return;
    }
    i = (*env)->GetIntField(env, obj, fid);
    printf("i = %d\n", i);
    (*env)->SetIntField(env, obj, fid, 2*i);
    printf("Ending the native method.\n");
}

Notice that this file includes jni.h, which contains interfacing information. This file is provided by your Java compiler. The header file NativeDemo.h was created by javah earlier.

In this function, the GetObjectClass() method is used to obtain a C structure that has information about the class NativeDemo. The GetFieldID() method returns a C structure with information about the field named “i” for the class. GetIntField() retrieves the original value of that field. SetIntField() stores an updated value in that field. (See the file jni.h for additional methods that handle other types of data.)
After creating NativeDemo.c, you must compile it and create a DLL. To do this by using the Microsoft C/C++ compiler, use the following command line. (You might need to specify the path to jni.h and its subordinate file jni_md.h.)

```sh
Cl /LD NativeDemo.c
```

This produces a file called NativeDemo.dll. Once this is done, you can execute the Java program, which will produce the following output:

```
This is ob.i before the native method: 10
Starting the native method.
i = 10
Ending the native method.
This is ob.i after the native method: 20
```

Problems with Native Methods

Native methods seem to offer great promise, because they enable you to gain access to an existing base of library routines, and they offer the possibility of faster run-time execution. But native methods also introduce two significant problems:

- Potential security risk Because a native method executes actual machine code, it can gain access to any part of the host system. That is, native code is not confined to the Java execution environment. This could allow a virus infection, for example. For this reason, applets cannot use native methods. Also, the loading of DLLs can be restricted, and their loading is subject to the approval of the security manager.
• Loss of portability Because the native code is contained in a DLL, it must be present on the machine that is executing the Java program. Further, because each native method is CPU- and operating system–dependent, each DLL is inherently nonportable. Thus, a Java application that uses native methods will be able to run only on a machine for which a compatible DLL has been installed.

The use of native methods should be restricted, because they render your Java programs nonportable and pose significant security risks.

Using assert
Another relatively new addition to Java is the keyword assert. It is used during program development to create an assertion, which is a condition that should be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an assert statement. At run time, if the condition is true, no other action takes place. However, if the condition is false, then an AssertionError is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

The assert keyword has two forms. The first is shown here:
Here, condition is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default AssertionError object is thrown.

The second form of assert is shown here:

```
assert condition: expr;
```

In this version, expr is a value that is passed to the AssertionError constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for expr, but any non-void expression is allowed as long as it defines a reasonable string conversion.

Here is an example that uses assert. It verifies that the return value of getnum( ) is positive.

```java
// Demonstrate assert.
class AssertDemo {
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;
    }

    public static void main(String args[])
    {
        int n;

        for(int i=0; i < 10; i++) {
            n = getnum();

            assert n > 0; // will fail when n is 0

            System.out.println("n is " + n);
        }
     }
}
```
To enable assertion checking at run time, you must specify the -ea option. For example, to enable assertions for AssertDemo, execute it using this line:

```
java -ea AssertDemo
```

After compiling and running as just described, the program creates the following output:

```
n is 3
n is 2
n is 1
Exception in thread "main" java.lang.AssertionError
  at AssertDemo.main(AssertDemo.java:17)
```

In `main()`, repeated calls are made to the method `getnum()`, which returns an integer value. The return value of `getnum()` is assigned to `n` and then tested using this `assert` statement:

```
assert n > 0; // will fail when n is 0
```

This statement will fail when `n` equals 0, which it will after the fourth call. When this happens, an exception is thrown.

As explained, you can specify the message displayed when an assertion fails. For example, if you substitute

```
assert n > 0 : "n is negative!";
```

for the assertion in the preceding program, then the following output will be generated:

```
n is 3
n is 2
n is 1
```
Exception in thread "main" java.lang.AssertionError: n is negative!
at AssertDemo.main(AssertDemo.java:17)

One important point to understand about assertions is that you must not rely on them to perform any action actually required by the program. The reason is that normally, released code will be run with assertions disabled. For example, consider this variation of the preceding program:

```java
// A poor way to use assert!!!
class AssertDemo {
    // get a random number generator
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;  
    }

    public static void main(String args[]) {
        int n = 0;

        for(int i=0; i < 10; i++) {
            assert (n = getnum()) > 0; // This is not a good idea!
            System.out.println("n is " + n);
        }
    }
}
```

In this version of the program, the call to getnum( ) is moved inside the assert statement. Although this works fine if assertions are enabled, it will cause a malfunction when assertions are disabled, because the call to getnum( ) will never be executed! In fact, n must now be initialized, because the compiler will recognize that it might not be assigned a value by the assert statement.
Assertions are a good addition to Java because they streamline the type of error checking that is common during development. For example, prior to assert, if you wanted to verify that \( n \) was positive in the preceding program, you had to use a sequence of code similar to this:

```java
if(n < 0) {
   System.out.println("n is negative!");
   return; // or throw an exception
}
```

With assert, you need only one line of code. Furthermore, you don’t have to remove the assert statements from your released code.

Assertion Enabling and Disabling Options

When executing code, you can disable all assertions by using the -da option. You can enable or disable a specific package (and all of its subpackages) by specifying its name followed by three periods after the -ea or -da option. For example, to enable assertions in a package called MyPack, use

```bash
-ea:MyPack...
```

To disable assertions in MyPack, use

```bash
-da:MyPack...
```

You can also specify a class with the -ea or -da option. For example, this enables AssertDemo individually:

```bash
-ea:AssertDemo
```

Static Import
Java includes a feature called static import that expands the capabilities of the import keyword. By following import with the keyword static, an import statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member.

To understand the usefulness of static import, let’s begin with an example that does not use it. The following program computes the hypotenuse of a right triangle. It uses two static methods from Java’s built-in math class Math, which is part of java.lang. The first is Math.pow(), which returns a value raised to a specified power. The second is Math.sqrt(), which returns the square root of its argument.

```java
// Compute the hypotenuse of a right triangle.
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;

        // Notice how sqrt() and pow() must be qualified by their class name, which is Math.
        hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));

        System.out.println("Given sides of lengths " + side1 + " and " + side2 + " the hypotenuse is " + hypot);
    }
}
```
Because pow( ) and sqrt( ) are static methods, they must be called through the use of their class’ name, Math. This results in a somewhat unwieldy hypotenuse calculation:

\[ \text{hypot} = \text{Math.sqrt}(\text{Math.pow}(\text{side1}, 2) + \text{Math.pow}(\text{side2}, 2)); \]

As this simple example illustrates, having to specify the class name each time pow( ) or sqrt( ) (or any of Java’s other math methods, such as sin( ), cos( ), and tan( )) is used can grow tedious.

You can eliminate the tedium of specifying the class name through the use of static import, as shown in the following version of the preceding program:

```java
// Use static import to bring sqrt() and pow() into view.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Compute the hypotenuse of a right triangle.
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;

        side1 = 3.0;
        side2 = 4.0;

        // Here, sqrt() and pow() can be called by themselves, without their class name.
        hypot = sqrt(pow(side1, 2) + pow(side2, 2));

        System.out.println("Given sides of lengths "+
           side1 + " and " + side2 + 
           " the hypotenuse is " +
           hypot);
    }
}
```

In this version, the names sqrt and pow are brought into
A common practice in Java programming is to use static import statements to simplify the code. Here are some examples:

```java
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

After these statements, it is no longer necessary to qualify `sqrt()` or `pow()` with their class name. Therefore, the hypotenuse calculation can more conveniently be specified, as shown here:

```java
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

As you can see, this form is considerably more readable.

There are two general forms of the import static statement. The first, which is used by the preceding example, brings into view a single name. Its general form is shown here:

```java
import static pkg.type-name.static-member-name;
```

Here, `type-name` is the name of a class or interface that contains the desired static member. Its full package name is specified by `pkg`. The name of the member is specified by `static-member-name`.

The second form of static import imports all static members of a given class or interface. Its general form is shown here:

```java
import static pkg.type-name.*;
```

If you will be using many static methods or fields defined by a class, then this form lets you bring them into view without having to specify each individually. Therefore, the preceding program could have used this
import static java.lang.Math.*;

Of course, static import is not limited just to the Math class or just to methods. For example, this brings the static field System.out into view:

import static java.lang.System.out;

After this statement, you can output to the console without having to qualify out with System, as shown here:

out.println("After importing System.out, you can use out directly.");

Whether importing System.out as just shown is a good idea is subject to debate. Although it does shorten the statement, it is no longer instantly clear to anyone reading the program that the out being referred to is System.out.

One other point: in addition to importing the static members of classes and interfaces defined by the Java API, you can also use static import to import the static members of classes and interfaces that you create.

As convenient as static import can be, it is important not to abuse it. Remember, the reason that Java organizes its libraries into packages is to avoid namespace collisions. When you import static members, you are bringing those members into the global namespace. Thus, you are increasing the potential for namespace conflicts and for the inadvertent hiding of
other names. If you are using a static member once or twice in the program, it’s best not to import it. Also, some static names, such as System.out, are so recognizable that you might not want to import them. Static import is designed for those situations in which you are using a static member repeatedly, such as when performing a series of mathematical computations. In essence, you should use, but not abuse, this feature.

Invoking Overloaded Constructors Through this( )
When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the this keyword. The general form is shown here:

```
    this(arg-list)
```

When this( ) is executed, the overloaded constructor that matches the parameter list specified by arg-list is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to this( ) must be the first statement within the constructor.

To understand how this( ) can be used, let’s work through a short example. First, consider the following class that does not use this( ):

```java
class MyClass {
    int a;
    int b;

    // initialize a and b individually
    MyClass(int i, int j) {
        a = i;
        b = j;
    }
}
```
This class contains three constructors, each of which initializes the values of `a` and `b`. The first is passed individual values for `a` and `b`. The second is passed just one value, which is assigned to both `a` and `b`. The third gives `a` and `b` default values of zero.

By using `this()`, it is possible to rewrite `MyClass` as shown here:

```java
class MyClass {
    int a;
    int b;

    // initialize `a` and `b` individually
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // initialize `a` and `b` to the same value
    MyClass(int i) {
        this(i, i); // invokes MyClass(i, i)
    }

    // give `a` and `b` default values of 0
    MyClass() {
        this(0); // invokes MyClass(0)
    }
}
```

In this version of `MyClass`, the only constructor that
actually assigns values to the a and b fields is 
MyClass(int, int). The other two constructors simply 
invoke that constructor (either directly or indirectly) 
through this( ). For example, consider what happens 
when this statement executes:

```java
MyClass mc = new MyClass(8);
```

The call to MyClass(8) causes this(8, 8) to be executed, 
which translates into a call to MyClass(8, 8), because this 
is the version of the MyClass constructor whose 
parameter list matches the arguments passed via this( ). 
Now, consider the following statement, which uses the 
default constructor:

```java
MyClass mc2 = new MyClass();
```

In this case, this(0) is called. This causes MyClass(0) to be 
invoked because it is the constructor with the matching 
parameter list. Of course, MyClass(0) then calls 
MyClass(0,0) as just described.

One reason why invoking overloaded constructors 
through this( ) can be useful is that it can prevent the 
unnecessary duplication of code. In many cases, reducing 
duplicate code decreases the time it takes to load your 
class because often the object code is smaller. This is 
especially important for programs delivered via the 
Internet in which load times are an issue. Using this( ) 
can also help structure your code when constructors 
contain a large amount of duplicate code.

However, you need to be careful. Constructors that call 
this( ) will execute a bit slower than those that contain
all of their initialization code inline. This is because the call and return mechanism used when the second constructor is invoked adds overhead. If your class will be used to create only a handful of objects, or if the constructors in the class that call this() will be seldom used, then this decrease in run-time performance is probably insignificant. However, if your class will be used to create a large number of objects (on the order of thousands) during program execution, then the negative impact of the increased overhead could be meaningful. Because object creation affects all users of your class, there will be cases in which you must carefully weigh the benefits of faster load time against the increased time it takes to create an object.

Here is another consideration: for very short constructors, such as those used by MyClass, there is often little difference in the size of the object code whether this() is used or not. (Actually, there are cases in which no reduction in the size of the object code is achieved.) This is because the bytecode that sets up and returns from the call to this() adds instructions to the object file. Therefore, in these types of situations, even though duplicate code is eliminated, using this() will not obtain significant savings in terms of load time. However, the added cost in terms of overhead to each object's construction will still be incurred. Therefore, this() is most applicable to constructors that contain large amounts of initialization code, not those that simply set the value of a handful of fields.

There are two restrictions you need to keep in mind
when using this(). First, you cannot use any instance variable of the constructor’s class in a call to this(). Second, you cannot use super() and this() in the same constructor because each must be the first statement in the constructor.
Since the original 1.0 release in 1995, many new features have been added to Java. The one that has had the most profound impact is generics. Introduced by JDK 5, generics changed Java in two important ways. First, it added a new syntactical element to the language. Second, it caused changes to many of the classes and methods in the core API. Today, generics are an integral part of Java programming, and a solid understanding of this important feature is required. It is examined here in detail.

Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data. Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type `Integer`, `String`, `Object`, or `Thread`. With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort. The expressive power generics added to the language fundamentally changed the way that Java code is written.

Perhaps the one feature of Java that has been most significantly affected by generics is the Collections Framework. The Collections Framework is part of the Java API and is described in detail in Chapter 17, but a brief mention is useful now. A collection is a group of
objects. The Collections Framework defines several classes, such as lists and maps, that manage collections. The collection classes have always been able to work with any type of object. The benefit that generics add is that the collection classes can now be used with complete type safety. Thus, in addition to being a powerful language element on its own, generics also enabled an existing feature to be substantially improved. This is another reason why generics represent such an important addition to Java.

This chapter describes the syntax, theory, and use of generics. It also shows how generics provide type safety for some previously difficult cases. Once you have completed this chapter, you will want to examine Chapter 17, which covers the Collections Framework. There you will find many examples of generics at work.

REMEMBER

Generics were added by JDK 5. Source code using generics cannot be compiled by earlier versions of javac.

What Are Generics?

At its core, the term generics means parameterized types. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.
It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type `Object`. Because `Object` is the superclass of all other classes, an `Object` reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used `Object` references to operate on various types of objects. The problem was that they could not do so with type safety.

Generics added the type safety that was lacking. They also streamlined the process, because it is no longer necessary to explicitly employ casts to translate between `Object` and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit. Thus, generics expanded your ability to reuse code and let you do so safely and easily.

**NOTE**

A Warning to C++ Programmers: Although generics are similar to templates in C++, they are not the same. There are some fundamental differences between the two approaches to generic types. If you have a background in C++, it is important not to jump to conclusions about how generics work in Java.

**A Simple Generics Example**

Let's begin with a simple example of a generic class. The following program defines two classes. The first is the generic class `Gen`, and the second is `GenDemo`, which uses `Gen`. 
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is created.
class Gen<T> {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }

    // Show type of T.
    void showType() {
        System.out.println("Type of T is "+
                         ob.getClass().getName());
    }
}

// Demonstrate the generic class.
class GenDemo {
    public static void main(String args[]) {
        // Create a Gen reference for Integers.
        Gen<Integer> iOb;

        // Create a Gen<Integer> object and assign its
        // reference to iOb. Notice the use of autoboxing
        // to encapsulate the value 88 within an Integer object.
        iOb = new Gen<Integer>(88);
The output produced by the program is shown here:

Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test

Let's examine this program carefully.

First, notice how Gen is declared by the following line:

```java
class Gen<T> {
```

Here, T is the name of a type parameter. This name is used as a placeholder for the actual type that will be passed to Gen when an object is created. Thus, T is used within Gen whenever the type parameter is needed. Notice that T is contained within < >. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets. Because
Gen uses a type parameter, Gen is a generic class, which is also called a parameterized type.

Next, T is used to declare an object called ob, as shown here:

```
T ob; // declare an object of type T
```

As explained, T is a placeholder for the actual type that will be specified when a Gen object is created. Thus, ob will be an object of the type passed to T. For example, if type String is passed to T, then in that instance, ob will be of type String.

Now consider Gen’s constructor:

```
Gen(T o) {
    ob = o;
}
```

Notice that its parameter, o, is of type T. This means that the actual type of o is determined by the type passed to T when a Gen object is created. Also, because both the parameter o and the member variable ob are of type T, they will both be of the same actual type when a Gen object is created.

The type parameter T can also be used to specify the return type of a method, as is the case with the getob() method, shown here:

```
T getob() {
    return ob;
}
```

Because ob is also of type T, its type is compatible with the return type specified by getob().

The showType() method displays the type of T by calling getName() on the Class object returned by the
call to getClass( ) on ob. The getClass( ) method is defined by Object and is thus a member of all class types. It returns a Class object that corresponds to the type of the class of the object on which it is called. Class defines the getName( ) method, which returns a string representation of the class name.

The GenDemo class demonstrates the generic Gen class. It first creates a version of Gen for integers, as shown here:

```java
Gen<Integer> iOb;
```

Look closely at this declaration. First, notice that the type Integer is specified within the angle brackets after Gen. In this case, Integer is a type argument that is passed to Gen’s type parameter, T. This effectively creates a version of Gen in which all references to T are translated into references to Integer. Thus, for this declaration, ob is of type Integer, and the return type of getob( ) is of type Integer.

Before moving on, it’s necessary to state that the Java compiler does not actually create different versions of Gen, or of any other generic class. Although it’s helpful to think in these terms, it is not what actually happens. Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code behave as if a specific version of Gen were created. Thus, there is really only one version of Gen that actually exists in your program. The process of removing generic type information is called erasure, and we will return to this topic later in this chapter.

The next line assigns to iOb a reference to an instance
of an Integer version of the Gen class:

```java
iOb = new Gen<Integer>(88);
```

Notice that when the Gen constructor is called, the type argument Integer is also specified. This is necessary because the type of the object (in this case iOb) to which the reference is being assigned is of type Gen<Integer>. Thus, the reference returned by new must also be of type Gen<Integer>. If it isn’t, a compile-time error will result. For example, the following assignment will cause a compile-time error:

```java
iOb = new Gen<Double>(88.0); // Error!
```

Because iOb is of type Gen<Integer>, it can’t be used to refer to an object of Gen<Double>. This type checking is one of the main benefits of generics because it ensures type safety.

As the comments in the program state, the assignment

```java
iOb = new Gen<Integer>(88);
```

makes use of autoboxing to encapsulate the value 88, which is an int, into an Integer. This works because Gen<Integer> creates a constructor that takes an Integer argument. Because an Integer is expected, Java will automatically box 88 inside one. Of course, the assignment could also have been written explicitly, like this:

```java
iOb = new Gen<Integer>(new Integer(88));
```

However, there would be no benefit to using this version.

The program then displays the type of ob within iOb,
which is Integer. Next, the program obtains the value of 
ob by use of the following line:

```java
int v = iOb.getob();
```

Because the return type of `getob()` is `T`, which was 
replaced by `Integer` when `iOb` was declared, the return 
type of `getob()` is also `Integer`, which unboxes into `int` 
when assigned to `v` (which is an `int`). Thus, there is no 
need to cast the return type of `getob()` to `Integer`. Of 
course, it’s not necessary to use the auto-unboxing 
feature. The preceding line could have been written like 
this, too:

```java
int v = iOb.getob().intValue();
```

However, the auto-unboxing feature makes the code 
more compact.

Next, `GenDemo` declares an object of type 
`Gen<String>`:

```java
Gen<String> strOb = new Gen<String>("Generics Test");
```

Because the type argument is `String`, `String` is substituted 
for `T` inside `Gen`. This creates (conceptually) a `String` 
version of `Gen`, as the remaining lines in the program 
demonstrate.

Generics Work Only with Objects

When declaring an instance of a generic type, the type 
argument passed to the type parameter must be a class 
type. You cannot use a primitive type, such as `int` or 
`char`. For example, with `Gen`, it is possible to pass any 
class type to `T`, but you cannot pass a primitive type to a 
type parameter. Therefore, the following declaration is
illegal:

```java
Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type
```

Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers (as the preceding example did) to encapsulate a primitive type. Further, Java’s autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

Generic Types Differ Based on Their Type Arguments

A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

```java
iOb = strOb; // Wrong!
```

Even though both iOb and strOb are of type Gen<T>, they are references to different types because their type parameters differ. This is part of the way that generics add type safety and prevent errors.

How Generics Improve Type Safety

At this point, you might be asking yourself the following question: Given that the same functionality found in the generic Gen class can be achieved without generics, by simply specifying Object as the data type and employing the proper casts, what is the benefit of making Gen generic? The answer is that generics automatically ensure the type safety of all operations involving Gen. In the
process, they eliminate the need for you to enter casts and to type-check code by hand.

To understand the benefits of generics, first consider the following program that creates a non-generic equivalent of Gen:

```java
// NonGen is functionally equivalent to Gen
// but does not use generics.
class NonGen {
    Object ob; // ob is now of type Object

    // Pass the constructor a reference to
    // an object of type Object
    NonGen(Object o) {
        ob = o;
    }

    // Return type Object.
    Object getOb() {
        return ob;
    }

    // Show type of ob.
    void showType() {
        System.out.println("Type of ob is "+
            ob.getClass().getName());
    }
}
```
There are several things of interest in this version. First, notice that NonGen replaces all uses of T with Object. This makes NonGen able to store any type of object, as can the generic version. However, it also prevents the Java compiler from having any real knowledge about the type of data actually stored in NonGen, which is bad for two reasons. First, explicit casts must be employed to retrieve the stored data. Second, many kinds of type mismatch errors cannot be
found until run time. Let’s look closely at each problem.

Notice this line:

```java
int v = (Integer) iOb.getob();
```

Because the return type of `getob()` is `Object`, the cast to `Integer` is necessary to enable that value to be auto-unboxed and stored in `v`. If you remove the cast, the program will not compile. With the generic version, this cast was implicit. In the non-generic version, the cast must be explicit. This is not only an inconvenience, but also a potential source of error.

Now, consider the following sequence from near the end of the program:

```java
// This compiles, but is conceptually wrong!
iOb = strOb;
v = (Integer) iOb.getob(); // run-time error!
```

Here, `strOb` is assigned to `iOb`. However, `strOb` refers to an object that contains a string, not an integer. This assignment is syntactically valid because all `NonGen` references are the same, and any `NonGen` reference can refer to any other `NonGen` object. However, the statement is semantically wrong, as the next line shows. Here, the return type of `getob()` is cast to `Integer`, and then an attempt is made to assign this value to `v`. The trouble is that `iOb` now refers to an object that stores a `String`, not an `Integer`. Unfortunately, without the use of generics, the Java compiler has no way to know this. Instead, a run-time exception occurs when the cast to `Integer` is attempted. As you know, it is extremely bad to have run-time exceptions occur in your code!

The preceding sequence can’t occur when generics are
used. If this sequence were attempted in the generic version of the program, the compiler would catch it and report an error, thus preventing a serious bug that results in a run-time exception. The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics. Although using Object references to create “generic” code has always been possible, that code was not type safe, and its misuse could result in run-time exceptions. Generics prevent this from occurring. In essence, through generics, run-time errors are converted into compile-time errors. This is a major advantage.

A Generic Class with Two Type Parameters
You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following TwoGen class is a variation of the Gen class that has two type parameters:
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V> {
    T ob1;
    V ob2;

    // Pass the constructor a reference to
    // an object of type T and an object of type V.
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    // Show types of T and V.
    void showTypes() {
        System.out.println("Type of T is " +
                          ob1.getClass().getName());

        System.out.println("Type of V is " +
                          ob2.getClass().getName());
    }

    T getOb1() {
        return ob1;
    }
}
The output from this program is shown here:

Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics

Notice how TwoGen is declared:

```java
class TwoGen<T, V> {
    // ... implementation ...
}
```

It specifies two type parameters: T and V, separated by a comma. Because it has two type parameters, two type arguments must be passed to TwoGen when an object is created, as shown next:

```java
TwoGen<Integer, String> tgObj =
    new TwoGen<Integer, String>(88, "Generics");
```

In this case, Integer is substituted for T, and String is
substituted for V.

Although the two type arguments differ in this example, it is possible for both types to be the same. For example, the following line of code is valid:

```java
TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

In this case, both T and V would be of type String. Of course, if the type arguments were always the same, then two type parameters would be unnecessary.

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

```java
class class-name <type-param-list> { // …
```

Here is the syntax for declaring a reference to a generic class:

```java
class-name <type-arg-list> var-name =
    new class-name <type-arg-list>(cons-arg-list);
```

Bounded Types

In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter. For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles. Thus, you want to
specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like this:

```java
// Stats attempts (unsuccessfully) to
// create a generic class that can compute
// the average of an array of numbers of
// any given type.
//
// The class contains an error!
class Stats<T> {
    T[] nums; // nums is an array of type T

    // Pass the constructor a reference to
    // an array of type T.
    Stats(T[] o) {
        nums = o;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue(); // Error!!!
        return sum / nums.length;
    }
}
```

In Stats, the `average()` method attempts to obtain the double version of each number in the `nums` array by calling `doubleValue()`. Because all numeric classes, such as `Integer` and `Double`, are subclasses of `Number`, and `Number` defines the `doubleValue()` method, this method is available to all numeric wrapper classes. The trouble is that the compiler has no way to know that you are intending to create `Stats` objects using only numeric types. Thus, when you try to compile `Stats`, an error is reported that indicates that the `doubleValue()` method is unknown. To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to `T`. Furthermore, you need some way to ensure
that only numeric types are actually passed.

To handle such situations, Java provides bounded types. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an extends clause when specifying the type parameter, as shown here:

```java
<T extends superclass>
```

This specifies that T can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit.

You can use an upper bound to fix the Stats class shown earlier by specifying Number as an upper bound, as shown here:
class Stats<T extends Number> {
    T[] nums; // array of Number or subclass

    // Pass the constructor a reference to
    // an array of type Number or subclass.
    Stats(T[] o) {
        nums = o;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();
        return sum / nums.length;
    }
}
// Demonstrate Stats.
class BoundsDemo {
    public static void main(String args[]) {

        Integer inums[] = { 1, 2, 3, 4, 5 };  
        Stats<Integer> iob = new Stats<Integer>(inums);  
        double v = iob.average();  
        System.out.println("iob average is " + v);

        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };  
        Stats<Double> dob = new Stats<Double>(dnums);  
        double w = dob.average();  
        System.out.println("dob average is " + w);

        // This won't compile because String is not a  
        // subclass of Number.  
        // String strs[] = { "1", "2", "3", "4", "5" };  
        // Stats<String> strob = new Stats<String>(strs);

        // double x = strob.average();  
        // System.out.println("strob average is " + v);
    }
}

The output is shown here:

Average is 3.0  
Average is 3.3

Notice how Stats is now declared by this line:

    class Stats<T extends Number> {

Because the type T is now bounded by Number, the Java compiler knows that all objects of type T can call doubleValue() because it is a method declared by Number. This is, by itself, a major advantage. However, as an added bonus, the bounding of T also prevents nonnumeric Stats objects from being created. For example, if you try removing the comments from the lines at the end of the program, and then try
recompiling, you will receive compile-time errors because String is not a subclass of Number.

In addition to using a class type as a bound, you can also use an interface type. In fact, you can specify multiple interfaces as bounds. Furthermore, a bound can include both a class type and one or more interfaces. In this case, the class type must be specified first. When a bound includes an interface type, only type arguments that implement that interface are legal. When specifying a bound that has a class and an interface, or multiple interfaces, use the & operator to connect them. For example,

```java
class Gen<T extends MyClass & MyInterface> { // …
```

Here, T is bounded by a class called MyClass and an interface called MyInterface. Thus, any type argument passed to T must be a subclass of MyClass and implement MyInterface.

Using Wildcard Arguments

As useful as type safety is, sometimes it can get in the way of perfectly acceptable constructs. For example, given the Stats class shown at the end of the preceding section, assume that you want to add a method called sameAvg( ) that determines if two Stats objects contain arrays that yield the same average, no matter what type of numeric data each object holds. For example, if one object contains the double values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same. One way to implement sameAvg( ) is to pass it a Stats argument, and then compare the average of that argument against the
invoking object, returning true only if the averages are the same. For example, you want to be able to call sameAvg( ), as shown here:

```java
Integer inums[] = { 1, 2, 3, 4, 5 };  
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };  

Stats<Integer> iob = new Stats<Integer>(inums);  
Stats<Double> dob = new Stats<Double>(dnums);  

if(iob.sameAvg(dob))  
    System.out.println("Averages are the same.");  
else  
    System.out.println("Averages differ.");
```

At first, creating sameAvg( ) seems like an easy problem. Because Stats is generic and its average( ) method can work on any type of Stats object, it seems that creating sameAvg( ) would be straightforward. Unfortunately, trouble starts as soon as you try to declare a parameter of type Stats. Because Stats is a parameterized type, what do you specify for Stats’ type parameter when you declare a parameter of that type?

At first, you might think of a solution like this, in which T is used as the type parameter:

```java
// This won't work!  
// Determine if two averages are the same.  
boolean sameAvg(Stats<T> ob) {  
    if(average() == ob.average())  
        return true;  
    return false;
}
```

The trouble with this attempt is that it will work only with other Stats objects whose type is the same as the invoking object. For example, if the invoking object is of type Stats<Integer>, then the parameter ob must also be of type Stats<Integer>. It can’t be used to compare
the average of an object of type Stats<Double> with the average of an object of type Stats<Short>, for example. Therefore, this approach won’t work except in a very narrow context and does not yield a general (that is, generic) solution.

To create a generic sameAvg( ) method, you must use another feature of Java generics: the wildcard argument. The wildcard argument is specified by the ?, and it represents an unknown type. Using a wildcard, here is one way to write the sameAvg( ) method:

```java
// Determine if two averages are the same.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
        return true;
    return false;
}
```

Here, Stats<?> matches any Stats object, allowing any two Stats objects to have their averages compared. The following program demonstrates this:
class Stats<T extends Number> {
    T[] nums; // array of Number or subclass

    // Pass the constructor a reference to
    // an array of type Number or subclass.
    Stats(T[] o) {
        nums = o;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;

        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();

        return sum / nums.length;
    }

    // Determine if two averages are the same.
    // Notice the use of the wildcard.
    boolean sameAvg(Stats<?> ob) {
        if(average() == ob.average())
            return true;

        return false;
    }
}
One last point: It is important to understand that the wildcard does not affect what type of Stats objects can be created. This is governed by the extends clause in the Stats declaration. The wildcard simply matches any valid Stats object.

Bounded Wildcards
Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy. To understand why, let’s work through an example. Consider the following hierarchy of classes that encapsulate coordinates:

```java
// Two-dimensional coordinates.
class TwoD {
    int x, y;

    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
    int z;

    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
    int t;

    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}
```

At the top of the hierarchy is TwoD, which encapsulates a two-dimensional, XY coordinate. TwoD is inherited by ThreeD, which adds a third dimension, creating an XYZ coordinate. ThreeD is inherited by FourD, which adds a fourth dimension (time), yielding a
four-dimensional coordinate.

Shown next is a generic class called Coords, which stores an array of coordinates:

```java
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}
```

Notice that Coords specifies a type parameter bounded by TwoD. This means that any array stored in a Coords object will contain objects of type TwoD or one of its subclasses.

Now, assume that you want to write a method that displays the X and Y coordinates for each element in the coords array of a Coords object. Because all types of Coords objects have at least two coordinates (X and Y), this is easy to do using a wildcard, as shown here:

```java
static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates:");
    for(int i=0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " +
                            c.coords[i].y);
    System.out.println();
}
```

Because Coords is a bounded generic type that specifies TwoD as an upper bound, all objects that can be used to create a Coords object will be arrays of type TwoD, or of classes derived from TwoD. Thus, showXY() can display the contents of any Coords object.

However, what if you want to create a method that displays the X, Y, and Z coordinates of a ThreeD or FourD object? The trouble is that not all Coords objects
will have three coordinates, because a Coords<TwoD> object will only have X and Y. Therefore, how do you write a method that displays the X, Y, and Z coordinates for Coords<ThreeD> and Coords<FourD> objects, while preventing that method from being used with Coords<TwoD> objects? The answer is the bounded wildcard argument.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an extends clause in much the same way it is used to create a bounded type.

Using a bounded wildcard, it is easy to create a method that displays the X, Y, and Z coordinates of a Coords object, if that object actually has those three coordinates. For example, the following showXYZ() method shows the X, Y, and Z coordinates of the elements stored in a Coords object, if those elements are actually of type ThreeD (or are derived from ThreeD):

```java
static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("X Y Z Coordinates:");
    for(int i=0; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + " " +
                            c.coords[i].y + " " +
                            c.coords[i].z);
    System.out.println();
}
```

Notice that an extends clause has been added to the wildcard in the declaration of parameter c. It states that the ? can match any type as long as it is ThreeD, or a class derived from ThreeD. Thus, the extends clause establishes an upper bound that the ? can match.
Because of this bound, showXYZ() can be called with references to objects of type Coords<ThreeD> or Coords<FourD>, but not with a reference of type Coords<TwoD>. Attempting to call showXZY() with a Coords<TwoD> reference results in a compile-time error, thus ensuring type safety.

Here is an entire program that demonstrates the actions of a bounded wildcard argument:

```java
// Bounded Wildcard arguments.

// Two-dimensional coordinates.
class TwoD {
    int x, y;

    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
    int z;

    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
    int t;

    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}
```
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}

// Demonstrate a bounded wildcard.
class BoundedWildcard {
    static void showXY(Coords<?>> c) {
        System.out.println("X Y Coordinates:");
        for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
                                c.coords[i].y);
        System.out.println();
    }

    static void showXYZ(Coords<? extends ThreeD> c) {
        System.out.println("X Y Z Coordinates:");
        for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
                                 c.coords[i].y + " " +
                                 c.coords[i].z);
        System.out.println();
    }

    static void showAll(Coords<? extends FourD> c) {
        System.out.println("X Y Z T Coordinates:");
        for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
                                 c.coords[i].y + " " +
                                 c.coords[i].z + " " +
                                 c.coords[i].t);
        System.out.println();
    }
}
public static void main(String args[]) {
    TwoD td[] = {
        new TwoD(0, 0),
        new TwoD(7, 9),
        new TwoD(18, 4),
        new TwoD(-1, -23)
    };

    Coords<TwoD> tdlocs = new Coords<TwoD>(td);

    System.out.println("Contents of tdlocs.");
    showXY(tdlocs); // OK, is a TwoD
    // showXYZ(tdlocs); // Error, not a ThreeD
    // showAll(tdlocs); // Error, not a FourD

    // Now, create some FourD objects.
    FourD fd[] = {
        new FourD(1, 2, 3, 4),
        new FourD(6, 8, 14, 8),
        new FourD(22, 9, 4, 9),
        new FourD(3, -2, -23, 17)
    };

    Coords<FourD> fdlocs = new Coords<FourD>(fd);

    System.out.println("Contents of fdlocs.");
    // These are all OK.
    showXY(fdlocs);
    showXYZ(fdlocs);
    showAll(fdlocs);
}

The output from the program is shown here:

Contents of tdlocs.
X Y Coordinates:
0 0
7 9
18 4
-1 -23

Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2
X Y Z Coordinates:
1 2 3
Notice these commented-out lines:

```java
// showXYZ(tdlocs); // Error, not a ThreeD
// showAll(tdlocs); // Error, not a FourD
```

Because `tdlocs` is a `Coords(TwoD)` object, it cannot be used to call `showXYZ()` or `showAll()` because bounded wildcard arguments in their declarations prevent it. To prove this to yourself, try removing the comment symbols, and then attempt to compile the program. You will receive compilation errors because of the type mismatches.

In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

```java
<? extends superclass>
```

where `superclass` is the name of the class that serves as the upper bound. Remember, this is an inclusive clause because the class forming the upper bound (that is, specified by `superclass`) is also within bounds.

You can also specify a lower bound for a wildcard by adding a `super` clause to a wildcard declaration. Here is its general form:

```java
<? super subclass>
```

In this case, only classes that are superclasses of `subclass`
are acceptable arguments. This is an exclusive clause, because it will not match the class specified by subclass.

Creating a Generic Method

As the preceding examples have shown, methods inside a generic class can make use of a class’ type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

Let’s begin with an example. The following program declares a non-generic class called GenMethDemo and a static generic method within that class called isIn( ). The isIn( ) method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

```java
// Demonstrate a simple generic method.
class GenMethDemo {

    // Determine if an object is in an array.
    static <T, V extends T> boolean isIn(T x, V[] y) {
        for(int i=0; i < y.length; i++)
            if(x.equals(y[i])) return true;
        return false;
    }

    public static void main(String args[]) {
        // Use isIn() on Integers.
        Integer nums[] = { 1, 2, 3, 4, 5 };

        if(isIn(2, nums))
            System.out.println("2 is in nums");

        if(!isIn(7, nums))
            System.out.println("7 is not in nums");
    }
}
```
System.out.println();

// Use isIn() on Strings.
String strs[] = { "one", "two", "three",
    "four", "five" };

if(isIn("two", strs))
    System.out.println("two is in strs");

if(!isIn("seven", strs))
    System.out.println("seven is not in strs");

// Oops! Won't compile! Types must be compatible.
//    if(isIn("two", nums))
//      System.out.println("two is in strs");
}
}

The output from the program is shown here:

2 is in nums
7 is not in nums
two is in strs
seven is not in strs

Let’s examine isIn( ) closely. First, notice how it is declared by this line:

static <T, V extends T> boolean isIn(T x, V[] y) {

The type parameters are declared before the return type of the method. Second, notice that the type V is upper-bounded by T. Thus, V must either be the same as type T, or a subclass of T. This relationship enforces that isIn( ) can be called only with arguments that are compatible with each other. Also notice that isIn( ) is static, enabling it to be called independently of any object. Understand, though, that generic methods can be either static or non-static. There is no restriction in this regard.

Now, notice how isIn( ) is called within main( ) by
use of the normal call syntax, without the need to specify type arguments. This is because the types of the arguments are automatically discerned, and the types of T and V are adjusted accordingly. For example, in the first call:

```java
if(isIn(2, nums))
```

the type of the first argument is Integer (due to autoboxing), which causes Integer to be substituted for T. The base type of the second argument is also Integer, which makes Integer a substitute for V, too.

In the second call, String types are used, and the types of T and V are replaced by String.

Now, notice the commented-out code, shown here:

```java
//    if(isIn("two", nums))
//      System.out.println("two is in strs");
```

If you remove the comments and then try to compile the program, you will receive an error. The reason is that the type parameter V is bounded by T in the extends clause in V’s declaration. This means that V must be either type T, or a subclass of T. In this case, the first argument is of type String, making T into String, but the second argument is of type Integer, which is not a subclass of String. This causes a compile-time type-mismatch error. This ability to enforce type safety is one of the most important advantages of generic methods.

The syntax used to create `isIn( )` can be generalized. Here is the syntax for a generic method:

```java
<type-param-list > ret-type meth-name (param-list) {
  // ...
```
In all cases, type-param-list is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Generic Constructors

It is also possible for constructors to be generic, even if their class is not. For example, consider the following short program:

```java
// Use a generic constructor.
class GenCons {
    private double val;

    <T extends Number> GenCons(T arg) {
        val = arg.doubleValue();
    }

    void showval() {
        System.out.println("val: " + val);
    }
}

class GenConsDemo {
    public static void main(String args[]) {
        GenCons test = new GenCons(100);
        GenCons test2 = new GenCons(123.5F);

        test.showval();
        test2.showval();
    }
}
```

The output is shown here:

```
val: 100.0
val: 123.5
```

Because GenCons( ) specifies a parameter of a generic type, which must be a subclass of Number, GenCons( ) can be called with any numeric type, including Integer, Float, or Double. Therefore, even though GenCons is not a generic class, its constructor is generic.
Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces. Generic interfaces are specified just like generic classes. Here is an example. It creates an interface called MinMax that declares the methods min( ) and max( ), which are expected to return the minimum and maximum value of some set of objects.
// A generic interface example.

// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}

// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
    T[] vals;

    MyClass(T[] o) { vals = o; }

    // Return the minimum value in vals.
    public T min() {
        T v = vals[0];
        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) < 0) v = vals[i];
        return v;
    }

    // Return the maximum value in vals.
    public T max() {
        T v = vals[0];
        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) > 0) v = vals[i];
        return v;
    }
}

class GenIFDemo {
    public static void main(String args[]) {
        Integer inums[] = {3, 6, 2, 8, 6};
        Character chs[] = {'b', 'r', 'p', 'w'};

        MyClass<Integer> iob = new MyClass<Integer>(inums);
        MyClass<Character> cob = new MyClass<Character>(chs);

        System.out.println("Max value in inums: " + iob.max());
        System.out.println("Min value in inums: " + iob.min());
        System.out.println("Max value in chs: " + cob.max());
        System.out.println("Min value in chs: " + cob.min());
    }
}

The output is shown here:

Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b

Although most aspects of this program should be easy to understand, a couple of key points need to be made.
First, notice that MinMax is declared like this:

```java
interface MinMax<T extends Comparable<T>> {
```

In general, a generic interface is declared in the same way as is a generic class. In this case, the type parameter is `T`, and its upper bound is `Comparable`, which is an interface defined by `java.lang`. A class that implements `Comparable` defines objects that can be ordered. Thus, requiring an upper bound of `Comparable` ensures that `MinMax` can be used only with objects that are capable of being compared. (See Chapter 16 for more information on `Comparable`.) Notice that `Comparable` is also generic. It takes a type parameter that specifies the type of the objects being compared.

Next, `MinMax` is implemented by `MyClass`. Notice the declaration of `MyClass`, shown here:

```java
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Pay special attention to the way that the type parameter `T` is declared by `MyClass` and then passed to `MinMax`. Because `MinMax` requires a type that implements `Comparable`, the implementing class (`MyClass` in this case) must specify the same bound. Furthermore, once this bound has been established, there is no need to specify it again in the `implements` clause. In fact, it would be wrong to do so. For example, this line is incorrect and won’t compile:

```java
// This is wrong!
class MyClass<T extends Comparable<T>> implements MinMax<T extends Comparable<T>> {
```

Once the type parameter has been established, it is simply passed to the interface without further
In general, if a class implements a generic interface, then that class must also be generic, at least to the extent that it takes a type parameter that is passed to the interface. For example, the following attempt to declare MyClass is in error:

```java
class MyClass implements MinMax<T> { // Wrong!
```

Because MyClass does not declare a type parameter, there is no way to pass one to MinMax. In this case, the identifier T is simply unknown, and the compiler reports an error. Of course, if a class implements a specific type of generic interface, such as shown here:

```java
class MyClass implements MinMax<Integer> { // OK
```

then the implementing class does not need to be generic.

The generic interface offers two benefits. First, it can be implemented for different types of data. Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented. In the MinMax example, only types that implement the Comparable interface can be passed to T.

Here is the generalized syntax for a generic interface:

```java
interface interface-name <type-param-list> { // ...
```

Here, type-param-list is a comma-separated list of type parameters. When a generic interface is implemented, you must specify the type arguments, as shown here:

```java
class class-name <type-param-list>
    implements interface-name <type-arg-list> { ...
```
Raw Types and Legacy Code

Because support for generics did not exist prior to JDK 5, it was necessary to provide some transition path from old, pre-generics code. At the time of this writing, there is still a large amount of pre-generics legacy code that must remain both functional and compatible with generics. Pre-generics code must be able to work with generics, and generic code must be able to work with pre-generics code.

To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a raw type for the class. This raw type is compatible with legacy code, which has no knowledge of generics. The main drawback to using the raw type is that the type safety of generics is lost.

Here is an example that shows a raw type in action:
// Demonstrate a raw type.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// Demonstrate raw type.
class RawDemo {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen object for Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");

    // Create a raw-type Gen object and give it
    // a Double value.
    Gen raw = new Gen(new Double(98.6));

    // Cast here is necessary because type is unknown.
    double d = (Double) raw.getob();
    System.out.println("value: " + d);

    // The use of a raw type can lead to run-time
    // exceptions. Here are some examples.

    // The following cast causes a run-time error!
    // int i = (Integer) raw.getob(); // run-time error

    // This assignment overrides type safety.
    strOb = raw; // OK, but potentially wrong
    // String str = strOb.getob(); // run-time error

    // This assignment also overrides type safety.
    raw = iOb; // OK, but potentially wrong
    // d = (Double) raw.getob(); // run-time error
  }
}
This program contains several interesting things. First, a raw type of the generic Gen class is created by the following declaration:

```java
Gen raw = new Gen(new Double(98.6));
```

Notice that no type arguments are specified. In essence, this creates a Gen object whose type T is replaced by Object.

A raw type is not type safe. Thus, a variable of a raw type can be assigned a reference to any type of Gen object. The reverse is also allowed; a variable of a specific Gen type can be assigned a reference to a raw Gen object. However, both operations are potentially unsafe because the type checking mechanism of generics is circumvented.

This lack of type safety is illustrated by the commented-out lines at the end of the program. Let's examine each case. First, consider the following situation:

```java
// int i = (Integer) raw.getob(); // run-time error
```

In this statement, the value of ob inside raw is obtained, and this value is cast to Integer. The trouble is that raw contains a Double value, not an integer value. However, this cannot be detected at compile time because the type of raw is unknown. Thus, this statement fails at run time.

The next sequence assigns to a strOb (a reference of type Gen<String>) a reference to a raw Gen object:

```java
strOb = raw; // OK, but potentially wrong
//    String str = strOb.getob(); // run-time error
```

The assignment, itself, is syntactically correct, but
questionable. Because \texttt{strOb} is of type \texttt{Gen<String>}, it is assumed to contain a \texttt{String}. However, after the assignment, the object referred to by \texttt{strOb} contains a \texttt{Double}. Thus, at run time, when an attempt is made to assign the contents of \texttt{strOb} to \texttt{str}, a run-time error results because \texttt{strOb} now contains a \texttt{Double}. Thus, the assignment of a raw reference to a generic reference bypasses the type-safety mechanism.

The following sequence inverts the preceding case:

\begin{verbatim}
raw = iOb; // OK, but potentially wrong
//    d = (Double) raw.getob(); // run-time error
\end{verbatim}

Here, a generic reference is assigned to a raw reference variable. Although this is syntactically correct, it can lead to problems, as illustrated by the second line. In this case, \texttt{raw} now refers to an object that contains an \texttt{Integer} object, but the cast assumes that it contains a \texttt{Double}. This error cannot be prevented at compile time. Rather, it causes a run-time error.

Because of the potential for danger inherent in raw types, \texttt{javac} displays unchecked warnings when a raw type is used in a way that might jeopardize type safety. In the preceding program, these lines generate unchecked warnings:

\begin{verbatim}
Gen raw = new Gen(new Double(98.6));
strOb = raw; // OK, but potentially wrong
\end{verbatim}

In the first line, it is the call to the \texttt{Gen} constructor without a type argument that causes the warning. In the second line, it is the assignment of a raw reference to a generic variable that generates the warning.
At first, you might think that this line should also generate an unchecked warning, but it does not:

```java
class Gen<T> {
    T ob;
    Gen(T o) {
        ob = o;
    }
    // Return ob.
}
```
T getob() {
    return ob;
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

In this hierarchy, Gen2 extends the generic class Gen. Notice how Gen2 is declared by the following line:

class Gen2<T> extends Gen<T> {

The type parameter T is specified by Gen2 and is also passed to Gen in the extends clause. This means that whatever type is passed to Gen2 will also be passed to Gen. For example, this declaration,

    Gen2<Integer> num = new Gen2<Integer>(100);

passes Integer as the type parameter to Gen. Thus, the ob inside the Gen portion of Gen2 will be of type Integer.

    Notice also that Gen2 does not use the type parameter T except to pass it to the Gen superclass. Thus, even if a subclass of a generic superclass would otherwise not need to be generic, it still must specify the type parameter(s) required by its generic superclass.

    Of course, a subclass is free to add its own type parameters, if needed. For example, here is a variation on the preceding hierarchy in which Gen2 adds a type parameter of its own:
// A subclass can add its own type parameters.
class Gen<T> {
    T ob; // declare an object of type T

    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T> {
    V ob2;

    Gen2(T o, V o2) {
        super(o);
        ob2 = o2;
    }

    V getob2() {
        return ob2;
    }
}

// Create an object of type Gen2.
class HierDemo {
    public static void main(String args[]) {

        // Create a Gen2 object for String and Integer.
        Gen2<String, Integer> x =
            new Gen2<String, Integer>("Value is: ", 99);

        System.out.print(x.getob());
        System.out.println(x.getob2());
    }
}

Notice the declaration of this version of Gen2, which is shown here:

    class Gen2<T, V> extends Gen<T> {

Here, T is the type passed to Gen, and V is the type that
Here, \( T \) is the type passed to \( \text{Gen} \), and \( V \) is the type specific to \( \text{Gen2} \). \( V \) is used to declare an object called \( \text{ob2} \), and as a return type for the method \( \text{getob2}() \). In \( \text{main}() \), a \( \text{Gen2} \) object is created in which type parameter \( T \) is \( \text{String} \), and type parameter \( V \) is \( \text{Integer} \).

The program displays the following, expected, result:

Value is: 99

A Generic Subclass

It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass. For example, consider this program:

```java
// A non-generic class can be the superclass
// of a generic subclass.

// A non-generic class.
class NonGen {
    int num;
    NonGen(int i) {
        num = i;
    }
    int getnum() {
        return num;
    }
}

// A generic subclass.
class Gen<T> extends NonGen {
    T ob; // declare an object of type T
    Gen(T o, int i) {
        super(i);
        ob = o;
    }
    // Return ob.
    T getob() {
        return ob;
    }
}
```
// Create a Gen object.
class HierDemo2 {
    public static void main(String args[]) {

        // Create a Gen object for String.
        Gen<String> w = new Gen<String>("Hello", 47);

        System.out.print(w.getob() + " ");
        System.out.println(w.getnum());
    }
}

The output from the program is shown here:

    Hello 47

In the program, notice how Gen inherits NonGen in the following declaration:

class Gen<T> extends NonGen {

Because NonGen is not generic, no type argument is specified. Thus, even though Gen declares the type parameter T, it is not needed by (nor can it be used by) NonGen. Thus, NonGen is inherited by Gen in the normal way. No special conditions apply.

Run-Time Type Comparisons Within a Generic Hierarchy

Recall the run-time type information operator instanceof that was described in Chapter 13. As explained, instanceof determines if an object is an instance of a class. It returns true if an object is of the specified type or can be cast to the specified type. The instanceof operator can be applied to objects of generic classes. The following class demonstrates some of the type compatibility implications of a generic hierarchy:
// Use the instanceof operator with a generic class hierarchy.
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getOb() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

// Demonstrate run-time type ID implications of generic class hierarchy.
class HierDemo3 {
    public static void main(String args[]) {
        // Create a Gen object for Integers.
        Gen<Integer> iOb = new Gen<Integer>(88);

        // Create a Gen2 object for Integers.
        Gen2<Integer> iOb2 = new Gen2<Integer>(99);

        // Create a Gen2 object for Strings.
        Gen2<String> strOb2 = new Gen2<String>("Generics Test");

        // See if iOb2 is some form of Gen2.
        if(iOb2 instanceof Gen2<?>)
            System.out.println("iOb2 is instance of Gen2");
The output from the program is shown here:

```
iOb2 is instance of Gen2
iOb2 is instance of Gen

strOb2 is instance of Gen2
strOb2 is instance of Gen

iOb is instance of Gen
```

In this program, Gen2 is a subclass of Gen, which is generic on type parameter T. In main( ), three objects are created. The first is iOb, which is an object of type Gen<Integer>. The second is iOb2, which is an instance of Gen2<Integer>. Finally, strOb2 is an object of type Gen2<String>.
Then, the program performs these instanceof tests on the type of iOb2:

```java
// See if iOb2 is some form of Gen2.
if(iOb2 instanceof Gen2<?>)
    System.out.println("iOb2 is instance of Gen2");

// See if iOb2 is some form of Gen.
if(iOb2 instanceof Gen<?>)
    System.out.println("iOb2 is instance of Gen");
```

As the output shows, both succeed. In the first test, iOb2 is checked against Gen2<?> . This test succeeds because it simply confirms that iOb2 is an object of some type of Gen2 object. The use of the wildcard enables instanceof to determine if iOb2 is an object of any type of Gen2. Next, iOb2 is tested against Gen<?>, the superclass type. This is also true because iOb2 is some form of Gen, the superclass. The next few lines in main( ) show the same sequence (and same results) for strOb2.

Next, iOb, which is an instance of Gen<Integer> (the superclass), is tested by these lines:

```java
// See if iOb is an instance of Gen2, which it is not.
if(iOb instanceof Gen2<?>)
    System.out.println("iOb is instance of Gen2");

// See if iOb is an instance of Gen, which it is.
if(iOb instanceof Gen<?>)
    System.out.println("iOb is instance of Gen");
```

The first if fails because iOb is not some type of Gen2 object. The second test succeeds because iOb is some type of Gen object.

Now, look closely at these commented-out lines:

```java
// The following can't be compiled because
// generic type info does not exist at run time.
//    if(iOb2 instanceof Gen2<Integer>)
//      System.out.println("iOb2 is instance of Gen2<Integer>");
```
As the comments indicate, these lines can’t be compiled because they attempt to compare iOb2 with a specific type of Gen2, in this case, Gen2<Integer>. Remember, there is no generic type information available at runtime. Therefore, there is no way for instanceof to know if iOb2 is an instance of Gen2<Integer> or not.

Casting
You can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same. For example, assuming the foregoing program, this cast is legal:

```java
(Gen<Integer>) iOb2 // legal
```

because iOb2 is an instance of Gen<Integer>. But, this cast:

```java
(Gen<Long>) iOb2 // illegal
```

is not legal because iOb2 is not an instance of Gen<Long>.

Overriding Methods in a Generic Class
A method in a generic class can be overridden just like any other method. For example, consider this program in which the method getob() is overridden:
The output is shown here:
As the output confirms, the overridden version of getob() is called for objects of type Gen2, but the superclass version is called for objects of type Gen.

Type Inference with Generics
Beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic type. To begin, consider the following generic class:

class MyClass<T, V> {
  T ob1;
  V ob2;

  MyClass(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }
  // …
}

Prior to JDK 7, to create an instance of MyClass, you would have needed to use a statement similar to the following:

    MyClass<Integer, String> mcOb =
      new MyClass<Integer, String>(98, "A String");

Here, the type arguments (which are Integer and String) are specified twice: first, when mcOb is declared, and second, when a MyClass instance is created via new. Since generics were introduced by JDK 5, this is the form required by all versions of Java prior to JDK 7. Although there is nothing wrong, per se, with this form, it is a bit more verbose than it needs to be. In the new clause, the type of the type arguments can be readily inferred from the type of mcOb; therefore, there is really no reason
that they need to be specified a second time. To address this situation, JDK 7 adds a syntactic element that lets you avoid the second specification.

In JDK 7, the preceding declaration can be rewritten as shown here:

```java
MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
```

Notice that the instance creation portion simply uses `<>`, which is an empty type argument list. This is referred to as the diamond operator. It tells the compiler to infer the type arguments needed by the constructor in the new expression. The principal advantage of this type-inference syntax is that it shortens what are sometimes quite long declaration statements.

The preceding can be generalized. When type inference is used, the declaration syntax for a generic reference and instance creation has this general form:

```
class-name <type-arg-list> var-name = new class-name<> (cons-arg-list);
```

Here, the type argument list of the constructor in the new clause is empty.

Type inference can also be applied to parameter passing. For example, if the following method is added to MyClass,

```java
boolean isSame(MyClass<T, V> o) {
    if(ob1 == o.ob1 && ob2 == o.ob2) return true;
    else return false;
}
```

then the following call is legal in JDK 7:

```java
if(mcOb.isSame(new MyClass<>(1, "test"))) System.out.println("Same");
```
In this case, the type arguments for the argument passed to isSame( ) can be inferred.

It is important to understand that type inference won’t work in all cases. For example, given the following class hierarchy,

```java
class A<T, V> {}
class B<T, V> extends A<T, V>{ }
```

the following declaration (which does not use type inference) is legal:

```java
MyClass<A<Integer, Long>, String> mcOb2 =
    new MyClass<A<Integer, Long>, String>(new B<Integer, Long>(), "Generics");
```

Here, because a base class reference can refer to a derived class object, it is legal for mcOb2 to refer to a MyClass object that has the type

```java
MyClass<B<Integer, Long>, String>
```

even though the reference has the type

```java
MyClass<A<Integer, Long>, String>
```

However, attempting to use type inference to shorten the line as shown here, won’t work:

```java
// Won't Work!
MyClass<A<Integer, Long>, String> mcOb2 =
    new MyClass<>(new B<Integer, Long>(), "Generics");
```

In this case, a type mismatch error is reported.

Because the type-inference syntax is new to JDK 7 and won’t work with older compilers, the examples in this book will continue to use the full syntax when declaring instances of generic classes. This way, the examples will work with any Java compiler that supports generics.
Using the full-length syntax also makes it very clear precisely what is being created, which is very important in example code shown in a book. However, in your own code, the use of the type-inference syntax will streamline your declarations.

Erasure

Usually, it is not necessary to know the details about how the Java compiler transforms your source code into object code. However, in the case of generics, some general understanding of the process is important because it explains why the generic features work as they do—and why their behavior is sometimes a bit surprising. For this reason, a brief discussion of how generics are implemented in Java is in order.

An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java. Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of erasure.

In general, here is how erasure works. When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which is Object if no explicit bound is specified, and then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments. The compiler also enforces this type
compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

To better understand how erasure works, consider the following two classes:

```java
// Here, T is bound by Object by default.
class Gen<T> {
    T ob; // here, T will be replaced by Object
    Gen(T o) {
        ob = o;
    }
    // Return ob.
    T getob() {
        return ob;
    }
}

// Here, T is bound by String.
class GenStr<T extends String> {
    T str; // here, T will be replaced by String
    GenStr(T o) {
        str = o;
    }
    T getstr() { return str; }
}
```

After these two classes are compiled, the T in Gen will be replaced by Object. The T in GenStr will be replaced by String. Within the code for Gen and GenStr, casts are employed to ensure proper typing. For example, this sequence:

```java
Gen<Integer> iOb = new Gen<Integer>(99);
int x = iOb.getob();
```

would be compiled as if it were written like this:

```java
Gen iOb = new Gen(99);
```
Because of erasure, some things work a bit differently than you might think. For example, consider this short program that creates two objects of the generic Gen class just shown:

```java
class GenTypeDemo {
    public static void main(String args[]) {
        Gen<Integer> iOb = new Gen<Integer>(99);
        Gen<Float> fOb = new Gen<Float>(102.2F);

        System.out.println(iOb.getClass().getName());
        System.out.println(fOb.getClass().getName());
    }
}
```

The output from this program is shown here:

```
Gen
Gen
```

As you can see, the types of both iOb and fOb are Gen, not the Gen<Integer> and Gen<Float> that you might have expected. Remember, all type parameters are erased during compilation. At run time, only raw types actually exist.

Bridge Methods

Occasionally, the compiler will need to add a bridge method to a class to handle situations in which the type erasure of an overriding method in a subclass does not produce the same erasure as the method in the superclass. In this case, a method is generated that uses the type erasure of the superclass, and this method calls the method that has the type erasure specified by the subclass. Of course, bridge methods only occur at the bytecode level, are not seen by you, and are not
Although bridge methods are not something that you will normally need to be concerned with, it is still instructive to see a situation in which one is generated. Consider the following program:

```java
// A situation that creates a bridge method.
class Gen<T> {
    T ob; // declare an object of type T
    // Pass the constructor a reference to
    // an object of type T.
    Gen(T o) {
        ob = o;
    }

    // Return ob.
    T getob() {
        return ob;
    }
}

// A subclass of Gen.
class Gen2 extends Gen<String> {

    Gen2(String o) {
        super(o);
    }

    // A String-specific override of getob().
    String getob() {
        System.out.print("You called String getob()");
        return ob;
    }
}

// Demonstrate a situation that requires a bridge method.
class BridgeDemo {
    public static void main(String args[]) {

        // Create a Gen2 object for Strings.
        Gen2 strOb2 = new Gen2("Generics Test");

        System.out.println(strOb2.getob());
    }
}
```

In the program, the subclass Gen2 extends Gen, but
does so using a String-specific version of Gen, as its declaration shows:

```java
class Gen2 extends Gen<String> {

Furthermore, inside Gen2, getob() is overridden with String specified as the return type:

```java
// A String-specific override of getob().
String getob() {
    System.out.print("You called String getob(): ");
    return ob;
}
```

All of this is perfectly acceptable. The only trouble is that because of type erasure, the expected form of getob() will be

```java
Object getob() { // …
```

To handle this problem, the compiler generates a bridge method with the preceding signature that calls the String version. Thus, if you examine the class file for Gen2 by using javap, you will see the following methods:

```java
class Gen2 extends Gen<java.lang.String> {
    Gen2(java.lang.String);
    java.lang.String getob();
    java.lang.Object getob(); // bridge method
}
```

As you can see, the bridge method has been included. (The comment was added by the author and not by javap, and the precise output you see may vary based on the version of Java that you are using.)

There is one last point to make about bridge methods. Notice that the only difference between the two getob() methods is their return type. Normally, this would cause an error, but because this does not occur in your source
code, it does not cause a problem and is handled correctly by the JVM.

Ambiguity Errors

The inclusion of generics gives rise to a new type of error that you must guard against: ambiguity. Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```java
// Ambiguity caused by erasure on overloaded methods.
class MyGenClass<T, V> {
    T ob1;
    V ob2;

    // ...

    // These two overloaded methods are ambiguous and will not compile.
    void set(T o) {
        ob1 = o;
    }

    void set(V o) {
        ob2 = o;
    }
}
```

Notice that `MyGenClass` declares two generic types: `T` and `V`. Inside `MyGenClass`, an attempt is made to overload `set()` based on parameters of type `T` and `V`. This looks reasonable because `T` and `V` appear to be different types. However, there are two ambiguity problems here.

First, as `MyGenClass` is written, there is no requirement that `T` and `V` actually be different types. For example, it is perfectly correct (in principle) to construct
a MyGenClass object as shown here:

    MyGenClass<String, String> obj = new MyGenClass<String, String>();

In this case, both T and V will be replaced by String. This makes both versions of set() identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of set() reduces both versions to the following:

    void set(Object o) { // …

Thus, the overloading of set() as attempted in MyGenClass is inherently ambiguous.

Ambiguity errors can be tricky to fix. For example, if you know that V will always be some type of String, you might try to fix MyGenClass by rewriting its declaration as shown here:

    class MyGenClass<T, V extends String> { // almost OK!

This change causes MyGenClass to compile, and you can even instantiate objects like the one shown here:

    MyGenClass<Integer, String> x = new MyGenClass<Integer, String>();

This works because Java can accurately determine which method to call. However, ambiguity returns when you try this line:

    MyGenClass<String, String> x = new MyGenClass<String, String>();

In this case, since both T and V are String, which version of set() is to be called? The call to set() is now ambiguous.

Frankly, in the preceding example, it would be much
better to use two separate method names, rather than trying to overload set(). Often, the solution to ambiguity involves the restructuring of the code, because ambiguity often means that you have a conceptual error in your design.

Some Generic Restrictions
There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays. Each is examined here.

Type Parameters Can’t Be Instantiated
It is not possible to create an instance of a type parameter. For example, consider this class:

```csharp
// Can't create an instance of T.
class Gen<T> {
    T ob;

    Gen() {
        ob = new T(); // Illegal!!!
    }
}
```

Here, it is illegal to attempt to create an instance of T. The reason should be easy to understand: because T does not exist at run time, how would the compiler know what type of object to create? Remember, erasure removes all type parameters during the compilation process.

Restrictions on Static Members
No static member can use a type parameter declared by the enclosing class. For example, both of the static members of this class are illegal:
class Wrong<T> {
    // Wrong, no static variables of type T.
    static T ob;

    // Wrong, no static method can use T.
    static T getob() {
        return ob;
    }
}

Although you can’t declare static members that use a type parameter declared by the enclosing class, you can declare static generic methods, which define their own type parameters, as was done earlier in this chapter.

Generic Array Restrictions
There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element type is a type parameter. Second, you cannot create an array of type-specific generic references. The following short program shows both situations:
As the program shows, it’s valid to declare a reference to an array of type $T$, as this line does:

```java
T vals[]; // OK
```

But, you cannot instantiate an array of $T$, as this commented-out line attempts:

```java
// vals = new T[10]; // can't create an array of T
```

The reason you can’t create an array of $T$ is that $T$ does not exist at run time, so there is no way for the compiler to know what type of array to actually create.

However, you can pass a reference to a type-compatible array to `Gen()` when an object is created and assign that reference to `vals`, as the program does in this
vals = nums; // OK to assign reference to existent array

This works because the array passed to Gen has a known type, which will be the same type as T at the time of object creation.

Inside main(), notice that you can’t declare an array of references to a specific generic type. That is, this line

// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

won’t compile. Arrays of specific generic types simply aren’t allowed, because they can lead to a loss of type safety.

You can create an array of references to a generic type if you use a wildcard, however, as shown here:

Gen<?> gens[] = new Gen<?>[10]; // OK

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

Generic Exception Restriction

A generic class cannot extend Throwable. This means that you cannot create generic exception classes.
CHAPTER 15
String Handling

CHAPTER 16
Exploring java.lang

CHAPTER 17
java.util Part 1: The Collections Framework

CHAPTER 18
java.util Part 2: More Utility Classes

CHAPTER 19
Input/Output: Exploring java.io

CHAPTER 20
Exploring NIO

CHAPTER 21
Networking

CHAPTER 22
The Applet Class

CHAPTER 23
Event Handling

CHAPTER 24
Introducing the AWT: Working with Windows, Graphics, and Text

CHAPTER 25
Using AWT Controls, Layout Managers, and Menus

CHAPTER 26
Images

CHAPTER 27
The Concurrency Utilities

CHAPTER 28
Regular Expressions and Other Packages
A brief overview of Java’s string handling was presented in Chapter 7. In this chapter, it is described in detail. As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike some other languages that implement strings as character arrays, Java implements strings as objects of type String.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when you create a String object, you are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new String object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in
which a modifiable string is desired, Java provides two options: StringBuffer and StringBuilder. Both hold strings that can be modified after they are created.

The String, StringBuffer, and StringBuilder classes are defined in java.lang. Thus, they are available to all programs automatically. All are declared final, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the CharSequence interface.

One last point: To say that the strings within objects of type String are unchangeable means that the contents of the String instance cannot be changed after it has been created. However, a variable declared as a String reference can be changed to point at some other String object at any time.

The String Constructors

The String class supports several constructors. To create an empty String, call the default constructor. For example,

```java
String s = new String();
```

will create an instance of String with no characters in it.

Frequently, you will want to create strings that have initial values. The String class provides a variety of constructors to handle this. To create a String initialized by an array of characters, use the constructor shown here:
Here is an example:

```java
char chars[] = { 'a', 'b', 'c' };  
String s = new String(chars);
```

This constructor initializes s with the string “abc”.

You can specify a subrange of a character array as an initializer using the following constructor:

```java
String(char chars[], int startIndex, int numChars)
```

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

```java
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };  
String s = new String(chars, 2, 3);
```

This initializes s with the characters cde.

You can construct a String object that contains the same character sequence as another String object using this constructor:

```java
String(String strObj)
```

Here, strObj is a String object. Consider this example:

```java
// Construct one String from another.

class MakeString {
    public static void main(String args[]) {
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);
        System.out.println(s1);
    }
}
```
System.out.println(s2);
}

The output from this program is as follows:

Java
Java

As you can see, s1 and s2 contain the same string.

Even though Java’s char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. Two forms are shown here:

String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)

Here, asciiChars specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

// Construct string from subset of char array.
class SubStringCons {
    public static void main(String args[]) {
        byte ascii[] = {65, 66, 67, 68, 69, 70 };

        String s1 = new String(ascii);
        System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}

This program generates the following output:

ABCDEF
CDE

Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However, you will often want to use the default encoding provided by the platform.

NOTE

The contents of the array are copied whenever you create a String object from an array. If you modify the contents of the array after you have created the string, the String will be unchanged.

You can construct a String from a StringBuffer by using the constructor shown here:

String(StringBuffer strBufObj)

You can construct a String from a StringBuilder by using this constructor:

String(StringBuilder strBuildObj)

The following constructor supports the extended Unicode character set:
String\(\langle\) int codePoints\[\], int startIndex, int numChars\(\rangle\)

Here, codePoints is an array that contains Unicode code points. The resulting string is constructed from the range that begins at startIndex and runs for numChars.

There are also constructors that let you specify a CharSet.

**NOTE**

A discussion of Unicode code points and how they are handled by Java is found in Chapter 16.

**String Length**

The length of a string is the number of characters that it contains. To obtain this value, call the length\(()\) method, shown here:

\[
\text{int length()}
\]

The following fragment prints “3”, since there are three characters in the string \(s\):

```java
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

**Special String Operations**

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new String instances from string literals, concatenation of
multiple String objects by use of the + operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals

The earlier examples showed how to explicitly create a String instance from an array of characters by using the new operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a String object. Thus, you can use a string literal to initialize a String object. For example, the following code fragment creates two equivalent strings:

```java
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc"; // use string literal
```

Because a String object is created for every string literal, you can use a string literal any place you can use a String object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the length( ) method on the string “abc”. As expected, it prints “3”.

```java
System.out.println("abc".length());
```

String Concatenation
In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings:

```java
String age = "9";
String s = "He is " + age + " years old."
System.out.println(s);
```

This displays the string “He is 9 years old.”

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them. Here is an example:

```java
// Using concatenation to prevent long lines.
class ConCat {
    public static void main(String args[]) {
        String longStr = "This could have been " +
                         "a very long line that would have " +
                         "wrapped around. But string concatenation " +
                         "prevents this."

        System.out.println(longStr);
    }
}
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```java
int age = 9;
```
String s = "He is " + age + " years old."
System.out.println(s);

In this case, age is an int rather than another String, but the output produced is the same as before. This is because the int value in age is automatically converted into its string representation within a String object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of String.

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

String s = "four: " + 2 + 2;
System.out.println(s);

This fragment displays

four: 22

rather than the

four: 4

that you probably expected. Here’s why. Operator precedence causes the concatenation of “four” with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

String s = "four: " + (2 + 2);

Now s contains the string “four: 4”.

String Conversion and toString( )

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method \texttt{valueOf()} defined by the \texttt{String} class. \texttt{valueOf()} is overloaded for all the primitive types and for type \texttt{Object}. For the primitive types, \texttt{valueOf()} returns a string that contains the human-readable equivalent of the value with which it is called. For objects, \texttt{valueOf()} calls the \texttt{toString()} method on the object. We will look more closely at \texttt{valueOf()} later in this chapter. Here, let's examine the \texttt{toString()} method, because it is the means by which you can determine the string representation for objects of classes that you create.

Every class implements \texttt{toString()} because it is defined by \texttt{Object}. However, the default implementation of \texttt{toString()} is seldom sufficient. For most important classes that you create, you will want to override \texttt{toString()} and provide your own string representations. Fortunately, this is easy to do. The \texttt{toString()} method has this general form:

\begin{verbatim}
String toString()
\end{verbatim}

To implement \texttt{toString()}, simply return a \texttt{String} object that contains the human-readable string that appropriately describes an object of your class.

By overriding \texttt{toString()} for classes that you create,
you allow them to be fully integrated into Java's programming environment. For example, they can be used in `print()` and `println()` statements and in concatenation expressions. The following program demonstrates this by overriding `toString()` for the `Box` class:

```java
// Override toString() for Box class.
class Box {
    double width;
    double height;
    double depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    public String toString() {
        return "Dimensions are " + width + " by " + depth + " by " + height + ".";
    }
}

class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        String s = "Box b: " + b; // concatenate Box object

        System.out.println(b); // convert Box to string
        System.out.println(s);
    }
}
```

The output of this program is shown here:

Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0

As you can see, `Box`'s `toString()` method is
automatically invoked when a Box object is used in a concatenation expression or in a call to println( ).

Character Extraction
The String class provides a number of ways in which characters can be extracted from a String object. Several are examined here. Although the characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt( )
To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method. It has this general form:

    char charAt(int where)

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location. For example,

    char ch;
    ch = "abc".charAt(1);

assigns the value b to ch.

getChars( )
If you need to extract more than one character at a time,
you can use the getChars() method. It has this general form:

```java
void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)
```

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd−1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

The following program demonstrates getChars():

```java
class getCharsDemo {
    public static void main(String args[]) {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];

        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Here is the output of this program:

```
demo
```

getBytes( )
There is an alternative to `getChars()` that stores the characters in an array of bytes. This method is called `getBytes()`, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

```java
byte[ ] getBytes()
```

Other forms of `getBytes()` are also available. `getBytes()` is most useful when you are exporting a `String` value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

`toCharArray()`

If you want to convert all the characters in a `String` object into a character array, the easiest way is to call `toCharArray()`. It returns an array of characters for the entire string. It has this general form:

```java
char[ ] toCharArray()
```

This function is provided as a convenience, since it is possible to use `getChars()` to achieve the same result.

**String Comparison**

The `String` class includes a number of methods that compare strings or substrings within strings. Several are examined here.
To compare two strings for equality, use `equals()`.

```java
boolean equals(Object str)
```

Here, `str` is the String object being compared with the invoking String object. It returns `true` if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call `equalsIgnoreCase()`. When it compares two strings, it considers `A-Z` to be the same as `a-z`. It has this general form:

```java
boolean equalsIgnoreCase(String str)
```

Here, `str` is the String object being compared with the invoking String object. It, too, returns `true` if the strings contain the same characters in the same order, and false otherwise.

Here is an example that demonstrates `equals()` and `equalsIgnoreCase()`:

```java
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";
        System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
        System.out.println(s1 + " equals " + s3 + " -> " + s1.equalsIgnoreCase(s3));
        System.out.println(s1 + " equals " + s4 + " -> " + s1.equalsIgnoreCase(s4));
    }
}
```
The output from the program is shown here:

Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true

regionMatches( )
The regionMatches( ) method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

boolean regionMatches(int startIndex, String str2,
                        int str2StartIndex,       int
                        numChars)

boolean regionMatches(boolean ignoreCase,
                       int startIndex, String str2,
                       int str2StartIndex,       int
                       numChars)

For both versions, startIndex specifies the index at which the region begins within the invoking String object. The String being compared is specified by str2. The index at which the comparison will start within str2
is specified by \texttt{str2StartIndex}. The length of the substring being compared is passed in \texttt{numChars}. In the second version, if \texttt{ignoreCase} is true, the case of the characters is ignored. Otherwise, case is significant.

\texttt{startsWith()} and \texttt{endsWith()}  

\texttt{String} defines two methods that are, more or less, specialized forms of \texttt{regionMatches( )}. The \texttt{startsWith( )} method determines whether a given \texttt{String} begins with a specified string. Conversely, \texttt{endsWith( )} determines whether the \texttt{String} in question ends with a specified string. They have the following general forms:

\begin{verbatim}
  boolean startsWith(String str)
  boolean endsWith(String str)
\end{verbatim}

Here, \texttt{str} is the \texttt{String} being tested. If the string matches, true is returned. Otherwise, false is returned. For example,

\begin{verbatim}
  "Foobar".endsWith("bar")
\end{verbatim}

and

\begin{verbatim}
  "Foobar".startsWith("Foo")
\end{verbatim}

are both true.

A second form of \texttt{startsWith( )}, shown here, lets you specify a starting point:

\begin{verbatim}
  boolean startsWith(String str, int startIndex)
\end{verbatim}

Here, \texttt{startIndex} specifies the index into the invoking
string at which point the search will begin. For example, 

"Foobar".startsWith("bar", 3)

returns true.

equals() Versus ==

It is important to understand that the equals() method and the == operator perform two different operations. As just explained, the equals() method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

```java
// equals() vs ==
class EqualsNotEqualTo {
    public static void main(String args[]) {
        String s1 = "Hello";
        String s2 = new String(s1);

        System.out.println(s1 + " equals " + s2 + " -> " +
                          s1.equals(s2));
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

The variable s1 refers to the String instance created by “Hello”. The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the
Hello equals Hello -> true
Hello == Hello -> false

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The method compareTo( ) serves this purpose. It is specified by the Comparable<T> interface, which String implements. It has this general form:

```java
int compareTo(String str)
```

Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than zero</td>
<td>The invoking string is less than str.</td>
</tr>
<tr>
<td>Greater than zero</td>
<td>The invoking string is greater than str.</td>
</tr>
<tr>
<td>Zero</td>
<td>The two strings are equal.</td>
</tr>
</tbody>
</table>

Here is a sample program that sorts an array of strings. The program uses compareTo( ) to determine sort ordering for a bubble sort:

```java
// A bubble sort for Strings.
class SortString {
    static String arr[] = {
        "Now", "is", "the", "time", "for", "all", "good", "men",
    };
```
public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
        for(int i = j + 1; i < arr.length; i++) {
            if(arr[i].compareTo(arr[j]) < 0) {
                String t = arr[j];
                arr[j] = arr[i];
                arr[i] = t;
            }
        }
        System.out.println(arr[j]);
    }
}

The output of this program is the list of words:

Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to

As you can see from the output of this example, compareTo( ) takes into account uppercase and lowercase letters. The word “Now” came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing
If you want to ignore case differences when comparing two strings, use `compareToIgnoreCase()`

```java
int compareToIgnoreCase(String str)
```

This method returns the same results as `compareTo()`, except that case differences are ignored. You might want to try substituting it into the previous program. After doing so, “Now” will no longer be first.

**Searching Strings**

The `String` class provides two methods that allow you to search a string for a specified character or substring:

- `indexOf()` Searches for the first occurrence of a character or substring.
- `lastIndexOf()` Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or −1 on failure.

To search for the first occurrence of a character, use

```java
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```java
int lastIndexOf(int ch)
```

Here, `ch` is the character being sought.

To search for the first or last occurrence of a substring, use

```java
int indexOf(String str)
```
int indexOf(String str)
int lastIndexOf(String str)

Here, str specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)

int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)

Here, startIndex specifies the index at which point the search begins. For indexOf(), the search runs from startIndex to the end of the string. For lastIndexOf(), the search runs from startIndex to zero.

The following example shows how to use the various index methods to search inside of a String:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
    public static void main(String args[]) {
        String s = "Now is the time for all good men " +
                     "to come to the aid of their country."
                    .getBytes("UTF-8");

        System.out.println(s);
        System.out.println("indexOf(t) = " +
                           s.indexOf('t'));
        System.out.println("lastIndexOf(t) = " +
                           s.lastIndexOf('t'));
        System.out.println("indexOf(the) = " +
                           s.indexOf("the")) ;
        System.out.println("lastIndexOf(the) = " +
                           s.lastIndexOf("the")) ;
        System.out.println("indexOf(t, 10) = " +
                           s.indexOf('t', 10)) ;
    }
}
```
Here is the output of this program:

Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55

Modifying a String
Because String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use a String method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

substring( )
You can extract a substring using substring( ). It has two forms. The first is

String substring(int startIndex)

Here, startIndex specifies the index at which the substring will begin. This form returns a copy of the
substring that begins at startIndex and runs to the end of
the invoking string.

The second form of substring( ) allows you to specify
both the beginning and ending index of the substring:

String substring(int startIndex, int endIndex)

Here, startIndex specifies the beginning index, and
endIndex specifies the stopping point. The string
returned contains all the characters from the beginning
index, up to, but not including, the ending index.

The following program uses substring( ) to replace all
instances of one substring with another within a string:

// Substring replacement.
class StringReplace {
    public static void main(String args[]) {
        String org = "This is a test. This is, too.";
        String search = "is";
        String sub = "was";
        String result = "";
        int i;

        do { // replace all matching substrings
            System.out.println(org);
            i = org.indexOf(search);
            if(i != -1) {
                result = org.substring(0, i);
                result = result + sub;
                result = result + org.substring(i + search.length());
                org = result;
            }
        } while(i != -1);
    }
}

The output from this program is shown here:

This is a test. This is, too.
You can concatenate two strings using `concat( )`, shown here:

```java
String concat(String str)
```

This method creates a new object that contains the invoking string with the contents of `str` appended to the end. `concat( )` performs the same function as `+`. For example,

```java
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string “onetwo” into `s2`. It generates the same result as the following sequence:

```java
String s1 = "one";
String s2 = s1 + "two";
```

The `replace( )` method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

```java
String replace(char original, char replacement)
```

Here, `original` specifies the character to be replaced by the character specified by `replacement`. The resulting string is returned. For example,
String s = "Hello".replace('l', 'w');

puts the string “Hewwo” into s.

The second form of replace( ) replaces one character sequence with another. It has this general form:

    String replace(CharSequence original, CharSequence replacement)

trim( )

The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

    String trim( )

Here is an example:

    String s = " Hello World ".trim();

This puts the string “Hello World” into s.

The trim( ) method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state’s capital. It uses trim( ) to remove any leading or trailing whitespace that may have inadvertently been entered by the user.
Data Conversion Using `valueOf()`

The `valueOf()` method converts data from its internal format into a human-readable form. It is a static method that is overloaded within `String` for all of Java’s built-in types so that each type can be converted properly into a string. `valueOf()` is also overloaded for type `Object`, so an object of any class type you create can also be used as an argument. (Recall that `Object` is a superclass for all classes.) Here are a few of its forms:

```
static String valueOf(double num)
```
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[])

As discussed earlier, valueOf( ) is called when a string representation of some other type of data is needed—for example, during concatenation operations. You can call this method directly with any data type and get a reasonable String representation. All of the simple types are converted to their common String representation. Any object that you pass to valueOf( ) will return the result of a call to the object’s toString( ) method. In fact, you could just call toString( ) directly and get the same result.

For most arrays, valueOf( ) returns a rather cryptic string, which indicates that it is an array of some type. For arrays of char, however, a String object is created that contains the characters in the char array. There is a special version of valueOf( ) that allows you to specify a subset of a char array. It has this general form:

    static String valueOf(char chars[ ], int startIndex, int numChars)

Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring.

Changing the Case of Characters Within a String
The method `toLowerCase()` converts all the characters in a string from uppercase to lowercase. The `toUpperCase()` method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

```java
String toLowerCase()
String toUpperCase()
```

Both methods return a `String` object that contains the uppercase or lowercase equivalent of the invoking `String`. The default locale governs the conversion in both cases.

Here is an example that uses `toLowerCase()` and `toUpperCase()`:

```java
// Demonstrate toUpperCase() and toLowerCase().
class ChangeCase {
    public static void main(String args[])
    {
        String s = "This is a test.";

        System.out.println("Original: " + s);

        String upper = s.toUpperCase();
        String lower = s.toLowerCase();

        System.out.println("Uppercase: " + upper);
        System.out.println("Lowercase: " + lower);
    }
}
```

The output produced by the program is shown here:

```
Original: This is a test.
```

One other point: Overloaded versions of toLowerCase() and toUpperCase() that let you specify a Locale object to govern the conversion are also supplied. Specifying the locale can be quite important in some cases and can help internationalize your application.

Additional String Methods
In addition to those methods discussed earlier, String has many other methods, including those summarized in the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int codePointAt(int i)</td>
<td>Returns the Unicode code point at the location specified by i.</td>
</tr>
<tr>
<td>int codePointBefore(int i)</td>
<td>Returns the Unicode code point at the location that precedes that specified by i.</td>
</tr>
<tr>
<td>int codePointCount(int start, int end)</td>
<td>Returns the number of code points in the portion of the invoking String that are between start and end–1.</td>
</tr>
<tr>
<td>boolean contains(CharSequence str)</td>
<td>Returns true if the invoking object contains the string specified by str. Returns false, otherwise.</td>
</tr>
<tr>
<td>boolean contentEquals(CharSequence str)</td>
<td>Returns true if the invoking string contains the same string as str. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean contentEquals(StringBuffer str)</td>
<td>Returns true if the invoking string contains the same string as str. Otherwise, returns false.</td>
</tr>
<tr>
<td>static String format(String fmtstr, Object ... args)</td>
<td>Returns a string formatted as specified by fmtstr. (See Chapter 18 for details on formatting.)</td>
</tr>
<tr>
<td>static String format(Locale loc, String fmtstr, Object ... args)</td>
<td>Returns a string formatted as specified by fmtstr. Formatting is governed by the locale specified by loc. (See Chapter 18 for details on formatting.)</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Returns true if the invoking string contains no characters and has a length of zero.</td>
</tr>
<tr>
<td>boolean matches(string regExp)</td>
<td>Returns true if the invoking string matches the regular expression passed in regExp. Otherwise, returns false.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>int offsetByCodePoints(int start, int num)</code></td>
<td>Returns the index within the invoking string that is <code>num</code> code points beyond the starting index specified by <code>start</code>.</td>
</tr>
<tr>
<td><code>String replaceFirst(String regExp, String newStr)</code></td>
<td>Returns a string in which the first substring that matches the regular expression specified by <code>regExp</code> is replaced by <code>newStr</code>.</td>
</tr>
<tr>
<td><code>String replaceAll(String regExp, String newStr)</code></td>
<td>Returns a string in which all substrings that match the regular expression specified by <code>regExp</code> are replaced by <code>newStr</code>.</td>
</tr>
<tr>
<td><code>String[] split(String regExp)</code></td>
<td>Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <code>regExp</code>.</td>
</tr>
<tr>
<td><code>String[] split(String regExp, int max)</code></td>
<td>Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in <code>regExp</code>. The number of pieces is specified by <code>max</code>. If <code>max</code> is negative, then the invoking string is fully decomposed. Otherwise, if <code>max</code> contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If <code>max</code> is zero, the invoking string is fully decomposed.</td>
</tr>
<tr>
<td><code>(CharSequence subSequence(int startIndex, int stopIndex)</code></td>
<td>Returns a substring of the invoking string, beginning at <code>startIndex</code> and stopping at <code>stopIndex</code>. This method is required by the <code>CharSequence</code> interface, which is implemented by <code>String</code>.</td>
</tr>
</tbody>
</table>

Notice that several of these methods work with regular expressions. Regular expressions are described in Chapter 28.

**StringBuffer**

`StringBuffer` is a peer class of `String` that provides much of the functionality of strings. As you know, `String` represents fixed-length, immutable character sequences. In contrast, `StringBuffer` represents growable and writable character sequences. `StringBuffer` may have characters and substrings inserted in the middle or appended to the end. `StringBuffer` will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to
allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

```java
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in `chars` and reserves room for 16 more characters.

length( ) and capacity( )

The current length of a StringBuffer can be found via the `length()` method, while the total allocated capacity can
be found through the capacity( ) method. They have the following general forms:

```java
int length( )
int capacity( )
```

Here is an example:

```java
// StringBuffer length vs. capacity.
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");

        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Here is the output of this program, which shows how StringBuffer reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

Since sb is initialized with the string “Hello” when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity( )

If you want to preallocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer. This is useful if you know in advance that you will be
appending a large number of small strings to a StringBuffer. \(\text{ensureCapacity( )}\) has this general form:

\[
\text{void ensureCapacity(int minCapacity)}
\]

Here, \(\text{minCapacity}\) specifies the minimum size of the buffer. (A buffer larger than \(\text{minCapacity}\) may be allocated for reasons of efficiency.)

\(\text{setLength( )}\)

To set the length of the string within a StringBuffer object, use \(\text{setLength( )}\). Its general form is shown here:

\[
\text{void setLength(int len)}
\]

Here, \(\text{len}\) specifies the length of the string. This value must be nonnegative.

When you increase the size of the string, null characters are added to the end. If you call \(\text{setLength( )}\) with a value less than the current value returned by \(\text{length( )}\), then the characters stored beyond the new length will be lost. The \(\text{setCharAtDemo}\) sample program in the following section uses \(\text{setLength( )}\) to shorten a StringBuffer.

\(\text{charAt( ) and setCharAt( )}\)

The value of a single character can be obtained from a StringBuffer via the \(\text{charAt( )}\) method. You can set the value of a character within a StringBuffer using \(\text{setCharAt( )}\). Their general forms are shown here:
char charAt(int where)
void setCharAt(int where, char ch)

For charAt( ), where specifies the index of the character being obtained. For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character. For both methods, where must be nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates charAt( ) and setCharAt( ):

// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("Hello");
      System.out.println("buffer before = " + sb);
      System.out.println("charAt(1) before = " + sb.charAt(1));
      sb.setCharAt(1, 'i');
      sb.setLength(2);
      System.out.println("buffer after = " + sb);
      System.out.println("charAt(1) after = " + sb.charAt(1));
   }
}

Here is the output generated by this program:

buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i

getChars( )
To copy a substring of a StringBuffer into an array, use the getChars( ) method. It has this general form:
void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

append()

The append() method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms:

    StringBuffer append(String str)
    StringBuffer append(int num)
    StringBuffer append(Object obj)

The string representation of each parameter is obtained, often by calling String.valueOf(). The result is appended to the current StringBuffer object. The buffer itself is returned by each version of append(). This allows subsequent calls to be chained together, as shown in the following example:
// Demonstrate append().
class appendDemo {
    public static void main(String args[]) {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);

        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
    }
}

The output of this example is shown here:

    a = 42!

insert()
The insert() method inserts one string into another. It is overloaded to accept values of all the primitive types, plus Strings, Objects, and CharSequences. Like append(), it obtains the string representation of the value it is called with. This string is then inserted into the invoking StringBuffer object. These are a few of its forms:

    StringBuffer insert(int index, String str)
    StringBuffer insert(int index, char ch)
    StringBuffer insert(int index, Object obj)

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

The following sample program inserts "like" between "I" and "Java":

    // Demonstrate insert().
class insertDemo {
        public static void main(String args[]) {
            StringBuffer sb = new StringBuffer("I Java!");

            sb.insert(1, "like");
            System.out.println(sb.toString());
        }
    }
sb.insert(2, "like ");
System.out.println(sb);
}
}

The output of this example is shown here:

I like Java!

reverse( )
You can reverse the characters within a StringBuffer object using reverse( ), shown here:

    StringBuffer reverse( )

This method returns the reverse of the object on which it was called. The following program demonstrates reverse( ):

    // Using reverse() to reverse a StringBuffer.
    class ReverseDemo {
        public static void main(String args[]) {
            StringBuffer s = new StringBuffer("abcdef");

            System.out.println(s);
            s.reverse();
            System.out.println(s);
        }
    }

Here is the output produced by the program:

abcdef
fedcba

delete( ) and deleteCharAt( )
You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ). These
methods are shown here:

```java
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

The `delete()` method deletes a sequence of characters from the invoking object. Here, `startIndex` specifies the index of the first character to remove, and `endIndex` specifies an index one past the last character to remove. Thus, the substring deleted runs from `startIndex` to `endIndex–1`. The resulting `StringBuffer` object is returned.

The `deleteCharAt()` method deletes the character at the index specified by `loc`. It returns the resulting `StringBuffer` object.

Here is a program that demonstrates the `delete()` and `deleteCharAt()` methods:

```java
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("This is a test.");

      sb.delete(4, 7);
      System.out.println("After delete: " + sb);

      sb.deleteCharAt(0);
      System.out.println("After deleteCharAt: " + sb);
   }
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

`replace()`
You can replace one set of characters with another set inside a StringBuffer object by calling replace( ). Its signature is shown here:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex–1 is replaced. The replacement string is passed in str. The resulting StringBuffer object is returned.

The following program demonstrates replace( ):

```
// Demonstrate replace()
class replaceDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");

        sb.replace(5, 7, "was");
        System.out.println("After replace: " + sb);
    }
}
```

Here is the output:

```
After replace: This was a test.
```

substring( )

You can obtain a portion of a StringBuffer by calling substring( ). It has the following two forms:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```
The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object. The second form returns the substring that starts at startIndex and runs through endIndex–1. These methods work just like those defined for String that were described earlier.

Additional StringBuffer Methods

In addition to those methods just described, StringBuffer supplies several others, including those summarized in the following table:
The following program demonstrates `indexOf()` and `lastIndexOf()`:

class IndexOfDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("one two one");
        int i;

        i = sb.indexOf("one");
        System.out.println("First index: ", i);

        i = sb.lastIndexOf("one");
        System.out.println("Last index: ", i);
    }
}
StringBuilder

Introduced by JDK 5, StringBuilder is a recent addition to Java’s string handling capabilities. StringBuilder is identical to StringBuffer except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of StringBuilder is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use StringBuffer rather than StringBuilder.
This chapter discusses those classes and interfaces defined by java.lang. As you know, java.lang is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of Java programming. It is Java’s most widely used package.

java.lang includes the following classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Class</th>
<th>Class</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Enum</td>
<td>Process</td>
<td>String</td>
</tr>
<tr>
<td>Byte</td>
<td>Float</td>
<td>ProcessBuilder</td>
<td>StringBuffer</td>
</tr>
<tr>
<td>Character</td>
<td>InheritableThreadLocal</td>
<td>ProcessBuilder.Redirect</td>
<td>StringBuilder</td>
</tr>
<tr>
<td>Character.Subset</td>
<td>Integer</td>
<td>Runtime</td>
<td>System</td>
</tr>
<tr>
<td>Character.UnicodeBlock</td>
<td>Long</td>
<td>RuntimePermission</td>
<td>Thread</td>
</tr>
<tr>
<td>Class</td>
<td>Math</td>
<td>SecurityManager</td>
<td>ThreadGroup</td>
</tr>
<tr>
<td>ClassLoader</td>
<td>Number</td>
<td>Short</td>
<td>ThreadLocal</td>
</tr>
<tr>
<td>ClassValue</td>
<td>Object</td>
<td>StackTraceElement</td>
<td>Throwable</td>
</tr>
<tr>
<td>Compiler</td>
<td>Package</td>
<td>StrictMath</td>
<td>Void</td>
</tr>
<tr>
<td>Double</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

java.lang defines the following interfaces:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Interface</th>
<th>Interface</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendable</td>
<td>Cloneable</td>
<td>Readable</td>
<td>Readable</td>
</tr>
<tr>
<td>AutoCloseable</td>
<td>Comparable</td>
<td>Runnable</td>
<td>Runnable</td>
</tr>
<tr>
<td>CharSequence</td>
<td>Iterable</td>
<td>Thread.UncaughtExceptionHandler</td>
<td>Thread.UncaughtExceptionHandler</td>
</tr>
</tbody>
</table>

Several of the classes contained in java.lang contain deprecated methods, most dating back to Java 1.0. These deprecated methods are still provided by Java to support an ever-shrinking pool of legacy code and are not recommended for new code. Most of the deprecations took place prior to JDK 7, and these deprecated methods are not discussed here.

Primitive Type Wrappers
As mentioned in Part I of this book, Java uses primitive types, such as int and char, for performance reasons. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. Also, there is no way for two methods to refer to the same instance of an int. At times, you will need to create an object representation for one of these primitive types. For example, there are collection classes discussed in Chapter 17 that deal only with objects; to store a primitive type in one of these classes, you need to wrap the primitive type in a class.

To address this need, Java provides classes that correspond to each of the primitive types. In essence, these classes encapsulate, or wrap, the primitive types within a class. Thus, they are commonly referred to as type wrappers. The type wrappers were introduced in Chapter 12. They are examined in detail here.

Number

The abstract class Number defines a superclass that is implemented by the classes that wrap the numeric types byte, short, int, long, float, and double. Number has abstract methods that return the value of the object in each of the different number formats. For example, doubleValue( ) returns the value as a double, floatValue( ) returns the value as a float, and so on. These methods are shown here:

```java
byte byteValue( )
double doubleValue( )
float floatValue( )
```
int intValue( )
long longValue( )
short shortValue( )

The values returned by these methods might be rounded. Truncation is also possible.

Number has concrete subclasses that hold explicit values of each numeric type: Double, Float, Byte, Short, Integer, and Long.

Double and Float
Double and Float are wrappers for floating-point values of type double and float, respectively. The constructors for Float are shown here:

    Float(double num)
    Float(float num)
    Float(String str) throws NumberFormatException

As you can see, Float objects can be constructed with values of type float or double. They can also be constructed from the string representation of a floating-point number.

The constructors for Double are shown here:

    Double(double num)
    Double(String str) throws NumberFormatException

Double objects can be constructed with a double value or a string containing a floating-point value.

The methods defined by Float are shown in Table 16-1. The methods defined by Double are shown in Table
Both `Float` and `Double` define the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MAX_EXPONENT</code></td>
<td>Maximum exponent</td>
</tr>
<tr>
<td><code>MAX_VALUE</code></td>
<td>Maximum positive value</td>
</tr>
<tr>
<td><code>MIN_EXPONENT</code></td>
<td>Minimum exponent</td>
</tr>
<tr>
<td><code>MIN_NORMAL</code></td>
<td>Minimum positive normal value</td>
</tr>
<tr>
<td><code>MIN_VALUE</code></td>
<td>Minimum positive value</td>
</tr>
<tr>
<td><code>NaN</code></td>
<td>Not a number</td>
</tr>
<tr>
<td><code>POSITIVE_INFINITY</code></td>
<td>Positive infinity</td>
</tr>
<tr>
<td><code>NEGATIVE_INFINITY</code></td>
<td>Negative infinity</td>
</tr>
<tr>
<td><code>SIZE</code></td>
<td>The bit width of the wrapped value</td>
</tr>
<tr>
<td><code>TYPE</code></td>
<td>The <code>Class</code> object for float or double</td>
</tr>
</tbody>
</table>

### Method Descriptions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte byteValue()</code></td>
<td>Returns the value of the invoking object as a byte.</td>
</tr>
<tr>
<td><code>static int compare(float num1, float num2)</code></td>
<td>Compares the values of <code>num1</code> and <code>num2</code>. Returns 0 if the values are equal. Returns a negative value if <code>num1</code> is less than <code>num2</code>. Returns a positive value if <code>num1</code> is greater than <code>num2</code>.</td>
</tr>
<tr>
<td><code>int compareTo(Float f)</code></td>
<td>Compares the numerical value of the invoking object with that of <code>f</code>. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.</td>
</tr>
<tr>
<td><code>double doubleValue()</code></td>
<td>Returns the value of the invoking object as a double.</td>
</tr>
<tr>
<td><code>boolean equals(Object FloatObj)</code></td>
<td>Returns <code>true</code> if the invoking <code>Float</code> object is equivalent to <code>FloatObj</code>. Otherwise, it returns <code>false</code>.</td>
</tr>
<tr>
<td><code>static int floatToIntBits(float num)</code></td>
<td>Returns the IEEE-compatible, single-precision bit pattern that corresponds to <code>num</code>.</td>
</tr>
<tr>
<td><code>static int floatToRawIntBits(float num)</code></td>
<td>Returns the IEEE-compatible single-precision bit pattern that corresponds to <code>num</code>. A NaN value is preserved.</td>
</tr>
<tr>
<td><code>float floatValue()</code></td>
<td>Returns the value of the invoking object as a float.</td>
</tr>
<tr>
<td><code>int hashCode()</code></td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td><code>static float intBitsToFloat(int num)</code></td>
<td>Returns <code>float</code> equivalent of the IEEE-compatible, single-precision bit pattern specified by <code>num</code>.</td>
</tr>
<tr>
<td><code>int intValue()</code></td>
<td>Returns the value of the invoking object as an <code>int</code>.</td>
</tr>
<tr>
<td>Method Description</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>boolean isInfinite()</code></td>
<td></td>
</tr>
<tr>
<td>Returns true if the invoking object contains an infinite value. Otherwise, it returns false.</td>
<td></td>
</tr>
<tr>
<td>static <code>boolean isInfinite(float num)</code></td>
<td></td>
</tr>
<tr>
<td>Returns true if <code>num</code> specifies an infinite value. Otherwise, it returns false.</td>
<td></td>
</tr>
<tr>
<td><code>boolean isNaN()</code></td>
<td></td>
</tr>
<tr>
<td>Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false.</td>
<td></td>
</tr>
<tr>
<td>static <code>boolean isNaN(float num)</code></td>
<td></td>
</tr>
<tr>
<td>Returns true if <code>num</code> specifies a value that is not a number. Otherwise, it returns false.</td>
<td></td>
</tr>
<tr>
<td><code>long longValue()</code></td>
<td></td>
</tr>
<tr>
<td>Returns the value of the invoking object as a long.</td>
<td></td>
</tr>
<tr>
<td>static <code>float parseInt(String str)</code></td>
<td></td>
</tr>
<tr>
<td>Returns the float equivalent of the number contained in the string specified by <code>str</code> using radix 10.</td>
<td></td>
</tr>
<tr>
<td><code>short shortValue()</code></td>
<td></td>
</tr>
<tr>
<td>Returns the value of the invoking object as a short.</td>
<td></td>
</tr>
<tr>
<td>static <code>String toHexString(float num)</code></td>
<td></td>
</tr>
<tr>
<td>Returns a string containing the value of <code>num</code> in hexadecimal format.</td>
<td></td>
</tr>
<tr>
<td><code>String toString()</code></td>
<td></td>
</tr>
<tr>
<td>Returns the string equivalent of the invoking object.</td>
<td></td>
</tr>
<tr>
<td>static <code>String toString(float num)</code></td>
<td></td>
</tr>
<tr>
<td>Returns the string equivalent of the value specified by <code>num</code>.</td>
<td></td>
</tr>
<tr>
<td>static <code>Float valueOf(float num)</code></td>
<td></td>
</tr>
<tr>
<td>Returns a Float object containing the value passed in <code>num</code>.</td>
<td></td>
</tr>
<tr>
<td>static <code>Float valueOf(String str)</code></td>
<td></td>
</tr>
<tr>
<td>Returns the Float object that contains the value specified by the string in <code>str</code>.</td>
<td></td>
</tr>
</tbody>
</table>

**Table 16-1 The Methods Defined by Float**
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte byteValue( )</td>
<td>Returns the value of the invoking object as a byte.</td>
</tr>
<tr>
<td>static int compare(double num1, double num2)</td>
<td>Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.</td>
</tr>
<tr>
<td>int compareTo(Double d)</td>
<td>Compares the numerical value of the invoking object with that of d. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.</td>
</tr>
<tr>
<td>static long doubleToLongBits(double num)</td>
<td>Returns the IEEE-compatible, double-precision bit pattern that corresponds to num.</td>
</tr>
<tr>
<td>static long doubleToRawLongBits(double num)</td>
<td>Returns the IEEE-compatible double-precision bit pattern that corresponds to num. A NaN value is preserved.</td>
</tr>
<tr>
<td>double doubleValue( )</td>
<td>Returns the value of the invoking object as a double.</td>
</tr>
<tr>
<td>boolean equals(Object DoubleObj)</td>
<td>Returns true if the invoking Double object is equivalent to DoubleObj. Otherwise, it returns false.</td>
</tr>
<tr>
<td>float floatValue( )</td>
<td>Returns the value of the invoking object as a float.</td>
</tr>
<tr>
<td>int hashCode( )</td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td>int intValue( )</td>
<td>Returns the value of the invoking object as an int.</td>
</tr>
<tr>
<td>boolean isInfinite( )</td>
<td>Returns true if the invoking object contains an infinite value. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isInfinite(double num)</td>
<td>Returns true if num specifies an infinite value. Otherwise, it returns false.</td>
</tr>
<tr>
<td>boolean isNaN( )</td>
<td>Returns true if the invoking object contains a value that is not a number. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isNaN(double num)</td>
<td>Returns true if num specifies a value that is not a number. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static double longBitsToDouble(long num)</td>
<td>Returns double equivalent of the IEEE-compatible, double-precision bit pattern specified by num.</td>
</tr>
<tr>
<td>long longValue( )</td>
<td>Returns the value of the invoking object as a long.</td>
</tr>
</tbody>
</table>
The following example creates two Double objects—one by using a double value and the other by passing a string that can be parsed as a double:

```java
class DoubleDemo {
   public static void main(String args[]) {
      Double d1 = new Double(3.14159);
      Double d2 = new Double("314159E-5");

      System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
   }
}
```

As you can see from the following output, both constructors created identical Double instances, as shown by the equals( ) method returning true:

```
3.14159 = 3.14159 -> true
```

### Understanding isInfinite( ) and isNaN( )

Float and Double provide the methods isInfinite( ) and isNaN( ), which help when manipulating two special double and float values. These methods test for two
unique values de ned by the IEEE oating-point
speci cation: in nity and NaN (not a number). isInfinite(
) returns true if the value being tested is in nitely large
or small in magnitude. isNaN( ) returns true if the value
being tested is not a number.
The following example creates two Double objects;
one is infinite, and the other is not a number:
// Demonstrate isInfinite() and isNaN()
class InfNaN {
public static void main(String args[]) {
Double d1 = new Double(1/0.);
Double d2 = new Double(0/0.);

System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
}
}

This program generates the following output:
Infinity: true, false
NaN: false, true

Byte, Short, Integer, and Long
The Byte, Short, Integer, and Long classes are wrappers
for byte, short, int, and long integer types, respectively.
Their constructors are shown here:
Byte(byte num)
Byte(String str) throws NumberFormatException
Short(short num)
Short(String str) throws NumberFormatException
Integer(int num)
Integer(String str) throws NumberFormatException


Long(long num)
Long(String str) throws NumberFormatException

As you can see, these objects can be constructed from numeric values or from strings that contain valid whole number values.

The methods defined by these classes are shown in Tables 16-3 through 16-6. As you can see, they define methods for parsing integers from strings and converting strings back into integers. Variants of these methods allow you to specify the radix, or numeric base, for conversion. Common radices are 2 for binary, 8 for octal, 10 for decimal, and 16 for hexadecimal.

The following constants are defined:

<table>
<thead>
<tr>
<th>MIN_VALUE</th>
<th>Minimum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_VALUE</td>
<td>Maximum value</td>
</tr>
<tr>
<td>SIZE</td>
<td>The bit width of the wrapped value</td>
</tr>
<tr>
<td>TYPE</td>
<td>The Class object for byte, short, int, or long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte byteValue( )</td>
<td>Returns the value of the invoking object as a byte.</td>
</tr>
<tr>
<td>static int compare(byte num1, byte num2)</td>
<td>Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2. (Added by JDK 7.)</td>
</tr>
<tr>
<td>int compareTo(Byte b)</td>
<td>Compares the numerical value of the invoking object with that of b. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.</td>
</tr>
<tr>
<td>static Byte decode(String str) throws NumberFormatException</td>
<td>Returns a Byte object that contains the value specified by the string in str.</td>
</tr>
<tr>
<td>double doubleValue( )</td>
<td>Returns the value of the invoking object as a double.</td>
</tr>
<tr>
<td>boolean equals(Object ByteObj)</td>
<td>Returns true if the invoking Byte object is equivalent to ByteObj. Otherwise, it returns false.</td>
</tr>
<tr>
<td>float floatValue( )</td>
<td>Returns the value of the invoking object as a float.</td>
</tr>
<tr>
<td>int hashCode( )</td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>int intValue()</td>
<td>Returns the value of the invoking object as an int.</td>
</tr>
<tr>
<td>long longValue()</td>
<td>Returns the value of the invoking object as a long.</td>
</tr>
<tr>
<td>static byte parseByte(String str)</td>
<td>Returns the byte equivalent of the number contained in the string specified by str using radix 10.</td>
</tr>
<tr>
<td>static byte parseByte(String str, int radix)</td>
<td>Returns the byte equivalent of the number contained in the string specified by str using the specified radix.</td>
</tr>
<tr>
<td>short shortValue()</td>
<td>Returns the value of the invoking object as a short.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a string that contains the decimal equivalent of the invoking object.</td>
</tr>
<tr>
<td>static String toString(byte num)</td>
<td>Returns a string that contains the decimal equivalent of num.</td>
</tr>
<tr>
<td>static Byte valueOf(byte num)</td>
<td>Returns a Byte object containing the value passed in num.</td>
</tr>
<tr>
<td>static Byte valueOf(String str)</td>
<td>Returns a Byte object that contains the value specified by the string in str.</td>
</tr>
<tr>
<td>static Byte valueOf(String str, int radix)</td>
<td>Returns a Byte object that contains the value specified by the string in str using the specified radix.</td>
</tr>
</tbody>
</table>

Table 16-3 The Methods Defined by Byte
<table>
<thead>
<tr>
<th>Method Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static short parseShort(String \textit{str}) throws NumberFormatException</td>
<td>Returns the \textit{short} equivalent of the number contained in the string specified by \textit{str} using radix 10.</td>
</tr>
<tr>
<td>static short parseShort(String \textit{str}, int \textit{radix}) throws NumberFormatException</td>
<td>Returns the \textit{short} equivalent of the number contained in the string specified by \textit{str} using the specified \textit{radix}.</td>
</tr>
<tr>
<td>static short reverseBytes(short \textit{num})</td>
<td>Exchanges the high- and low-order bytes of \textit{num} and returns the result.</td>
</tr>
<tr>
<td>short shortValue()</td>
<td>Returns the value of the invoking object as a \textbf{short}.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a string that contains the decimal equivalent of the invoking object.</td>
</tr>
<tr>
<td>static String toString(short \textit{num})</td>
<td>Returns a string that contains the decimal equivalent of \textit{num}.</td>
</tr>
<tr>
<td>static Short valueOf(short \textit{num})</td>
<td>Returns a \textbf{Short} object containing the value passed in \textit{num}.</td>
</tr>
<tr>
<td>static Short valueOf(String \textit{str}) throws NumberFormatException</td>
<td>Returns a \textbf{Short} object that contains the value specified by the string in \textit{str} using radix 10.</td>
</tr>
<tr>
<td>static Short valueOf(String \textit{str}, int \textit{radix}) throws NumberFormatException</td>
<td>Returns a \textbf{Short} object that contains the value specified by the string in \textit{str} using the specified \textit{radix}.</td>
</tr>
</tbody>
</table>

Table 16-4 The Methods Defined by \textit{Short}
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int bitCount(int num)</td>
<td>Returns the number of set bits in num.</td>
</tr>
<tr>
<td>byte byteValue( )</td>
<td>Returns the value of the invoking object as a byte.</td>
</tr>
<tr>
<td>static int compare(int num1, int num2)</td>
<td>Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2. (Added by JDK 7.)</td>
</tr>
<tr>
<td>int compareTo(Integer i)</td>
<td>Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.</td>
</tr>
<tr>
<td>static Integer decode(String str) throws NumberFormatException</td>
<td>Returns an Integer object that contains the value specified by the string in str.</td>
</tr>
<tr>
<td>double doubleValue( )</td>
<td>Returns the value of the invoking object as a double.</td>
</tr>
<tr>
<td>boolean equals(Object IntegerObj)</td>
<td>Returns true if the invoking Integer object is equivalent to IntegerObj. Otherwise, it returns false.</td>
</tr>
<tr>
<td>float floatValue( )</td>
<td>Returns the value of the invoking object as a float.</td>
</tr>
<tr>
<td>static Integer getInteger(String propertyName)</td>
<td>Returns the value associated with the environmental property specified by propertyName. A null is returned on failure.</td>
</tr>
<tr>
<td>static Integer getInteger(String propertyName, int default)</td>
<td>Returns the value associated with the environmental property specified by propertyName. The value of default is returned on failure.</td>
</tr>
<tr>
<td>static Integer getInteger(String propertyName, Integer default)</td>
<td>Returns the value associated with the environmental property specified by propertyName. The value of default is returned on failure.</td>
</tr>
<tr>
<td>int hashCode( )</td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td>static int highestOneBit(int num)</td>
<td>Determines the position of the highest order set bit in num. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.</td>
</tr>
<tr>
<td>int intValue( )</td>
<td>Returns the value of the invoking object as an int.</td>
</tr>
<tr>
<td>long longValue( )</td>
<td>Returns the value of the invoking object as a long.</td>
</tr>
<tr>
<td>static int lowestOneBit(int num)</td>
<td>Determines the position of the lowest order set bit in num. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.</td>
</tr>
<tr>
<td>static int numberOfLeadingZeros(int num)</td>
<td>Returns the number of high-order zero bits that precede the first high-order set bit in num. If num is zero, 32 is returned.</td>
</tr>
<tr>
<td>static int numberOfTrailingZeros(int num)</td>
<td>Returns the number of low-order zero bits that precede the first low-order set bit in num. If num is zero, 32 is returned.</td>
</tr>
<tr>
<td>static int parseInt(String str) throws NumberFormatException</td>
<td>Returns the integer equivalent of the number contained in the string specified by str using radix 10.</td>
</tr>
<tr>
<td>static int parseInt(String str, int radix) throws NumberFormatException</td>
<td>Returns the integer equivalent of the number contained in the string specified by str using the specified radix.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>static int reverse(int num)</td>
<td>Reverses the order of the bits in num and returns the result.</td>
</tr>
<tr>
<td>static int reverseBytes(int num)</td>
<td>Reverses the order of the bytes in num and returns the result.</td>
</tr>
<tr>
<td>static int rotateLeft(int num, int n)</td>
<td>Returns the result of rotating num left n positions.</td>
</tr>
<tr>
<td>static int rotateRight(int num, int n)</td>
<td>Returns the result of rotating num right n positions.</td>
</tr>
<tr>
<td>static int signum(int num)</td>
<td>Returns -1 if num is negative, 0 if it is zero, and 1 if it is positive.</td>
</tr>
<tr>
<td>short shortValue( )</td>
<td>Returns the value of the invoking object as a short.</td>
</tr>
<tr>
<td>static String toBinaryString(int num)</td>
<td>Returns a string that contains the binary equivalent of num.</td>
</tr>
<tr>
<td>static String toHexString(int num)</td>
<td>Returns a string that contains the hexadecimal equivalent of num.</td>
</tr>
<tr>
<td>static String toOctalString(int num)</td>
<td>Returns a string that contains the octal equivalent of num.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns a string that contains the decimal equivalent of the invoking object.</td>
</tr>
<tr>
<td>static String toString(int num)</td>
<td>Returns a string that contains the decimal equivalent of num.</td>
</tr>
<tr>
<td>static String toString(int num, int radix)</td>
<td>Returns a string that contains the decimal equivalent of num using the specified radix.</td>
</tr>
<tr>
<td>static Integer valueOf(int num)</td>
<td>Returns an Integer object containing the value passed in num.</td>
</tr>
<tr>
<td>static Integer valueOf(String str)</td>
<td>Returns an Integer object that contains the value specified by the string in str.</td>
</tr>
<tr>
<td>static Integer valueOf(String str, int radix)</td>
<td>Returns an Integer object that contains the value specified by the string in str using the specified radix.</td>
</tr>
</tbody>
</table>

Table 16-5 The Methods Defined by Integer
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int bitCount(long num)</td>
<td>Returns the number of set bits in num.</td>
</tr>
<tr>
<td>byte byteValue( )</td>
<td>Returns the value of the invoking object as a byte.</td>
</tr>
<tr>
<td>static int compare(long num1, long num2)</td>
<td>Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2. (Added by JDK 7.)</td>
</tr>
<tr>
<td>int compareTo(Long l)</td>
<td>Compares the numerical value of the invoking object with that of l. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.</td>
</tr>
<tr>
<td>static Long decode(String str) throws NumberFormatException</td>
<td>Returns a Long object that contains the value specified by the string in str.</td>
</tr>
<tr>
<td>double doubleValue( )</td>
<td>Returns the value of the invoking object as a double.</td>
</tr>
<tr>
<td>boolean equals(Object LongObj)</td>
<td>Returns true if the invoking Long object is equivalent to LongObj. Otherwise, it returns false.</td>
</tr>
<tr>
<td>float floatValue( )</td>
<td>Returns the value of the invoking object as a float.</td>
</tr>
<tr>
<td>static Long getLong(String propertyName)</td>
<td>Returns the value associated with the environmental property specified by propertyName. A null is returned on failure.</td>
</tr>
<tr>
<td>static Long getLong(String propertyName, long default)</td>
<td>Returns the value associated with the environmental property specified by propertyName. The value of default is returned on failure.</td>
</tr>
<tr>
<td>static Long getLong(String propertyName, Long default)</td>
<td>Returns the value associated with the environmental property specified by propertyName. The value of default is returned on failure.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td>static long highestOneBit(long num)</td>
<td>Determines the position of the highest-order set bit in num. It returns a</td>
</tr>
</tbody>
</table>
|                                            | value in which only this bit is set. If no bit is set to one, then zero is 
|                                            | returned.                                                                   |
| int intValue()                              | Returns the value of the invoking object as an int.                         |
| long longValue()                            | Returns the value of the invoking object as a long.                         |
| static long lowestOneBit(long num)          | Determines the position of the lowest-order set bit in num. It returns a    |
|                                            | value in which only this bit is set. If no bit is set to one, then zero is 
|                                            | returned.                                                                   |
| static int numberOfLeadingZeros(long num)   | Returns the number of high-order zero bits that precede the first high-order|
|                                            | set bit in num. If num is zero, 64 is returned.                            |
| static int numberOfTrailingZeros(long num)  | Returns the number of low-order zero bits that precede the first low-order  |
|                                            | set bit in num. If num is zero, 64 is returned.                            |
| static long parseLong(String str)           | Returns the long equivalent of the number contained in the string specified  |
| throws NumberFormatException                | by str using radix 10.                                                      |
| static long parseLong(String str, int radix)| Returns the long equivalent of the number contained in the string specified  |
| throws NumberFormatException                | by str using the specified radix.                                           |
| static long reverse(long num)               | Reverses the order of the bits in num and returns the result.              |
| static long reverseBytes(long num)          | Reverses the order of the bytes in num and returns the result.             |
| static long rotateLeft(long num, int n)     | Returns the result of rotating num left n positions.                       |
| static long rotateRight(long num, int n)    | Returns the result of rotating num right n positions.                      |
Converting Numbers to and from Strings

One of the most common programming chores is converting the string representation of a number into its internal, binary format. Fortunately, Java provides an easy way to accomplish this. The Byte, Short, Integer, and Long classes provide the parseByte( ), parseShort( ), parseInt( ), and parseLong( ) methods, respectively. These methods return the byte, short, int, or long equivalent of the numeric string with which they are called. (Similar methods also exist for the Float and Double classes.)

The following program demonstrates parseInt( ). It sums a list of integers entered by the user. It reads the integers using readLine( ) and uses parseInt( ) to convert
These strings into their int equivalents.

/* This program sums a list of numbers entered by the user. It converts the string representation of each number into an int using parseInt(). */

import java.io.*;

class ParseDemo {
    public static void main(String args[])
        throws IOException
    {
        // create a BufferedReader using System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;
        int i;
        int sum=0;

        System.out.println("Enter numbers, 0 to quit.");
        do {
            str = br.readLine();
            try {
                i = Integer.parseInt(str);
            } catch(NumberFormatException e) {
                System.out.println("Invalid format");
                i = 0;
            }
            sum += i;
            System.out.println("Current sum is: " + sum);
        } while(i != 0);
    }
}

To convert a whole number into a decimal string, use the versions of toString( ) defined in the Byte, Short, Integer, or Long classes. The Integer and Long classes also provide the methods toBinaryString( ), toHexString( ), and toOctalString( ), which convert a value into a binary, hexadecimal, or octal string, respectively.

The following program demonstrates binary, hexadecimal, and octal conversion:
class StringConversions {
    public static void main(String args[]) {
        int num = 19648;
        System.out.println(num + " in binary: " +
            Integer.toBinaryString(num));

        System.out.println(num + " in octal: " +
            Integer.toOctalString(num));

        System.out.println(num + " in hexadecimal: " +
            Integer.toHexString(num));
    }
}

The output of this program is shown here:

19648 in binary: 100110011000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0

Character

Character is a simple wrapper around a char. The constructor for Character is

    Character(char ch)

Here, ch specifies the character that will be wrapped by the Character object being created.

To obtain the char value contained in a Character object, call charValue( ), shown here:

    char charValue( )

It returns the character.

The Character class defines several constants, including the following:
Character includes several static methods that categorize characters and alter their case. They are shown in Table 16-7. The following example demonstrates several of these methods:

```java
// Demonstrate several Is… methods.

class IsDemo {
    public static void main(String args[]) {
        char a[] = {'a', 'b', '5', '?', 'A', ' '};

        for(int i=0; i<a.length; i++) {
            if(Character.isDigit(a[i]))
                System.out.println(a[i] + " is a digit.");
            if(Character.isLetter(a[i]))
                System.out.println(a[i] + " is a letter.");
            if(Character.isWhitespace(a[i]))
                System.out.println(a[i] + " is whitespace.");
            if(Character.isUpperCase(a[i]))
                System.out.println(a[i] + " is uppercase.");
            if(Character.isLowerCase(a[i]))
                System.out.println(a[i] + " is lowercase.");
        }
    }
}
```

The output from this program is shown here:

```
a is a letter.
a is lowercase.
b is a letter.
b is lowercase.
5 is a digit.
A is a letter.
A is uppercase.
is whitespace.
```

Character defines two methods, forDigit( ) and digit( ),
that enable you to convert between integer values and the digits they represent. They are shown here:

```java
static char forDigit(int num, int radix)
static int digit(char digit, int radix)
```

forDigit() returns the digit character associated with the value of num. The radix of the conversion is specified by radix. digit() returns the integer value associated with the specified character (which is presumably a digit) according to the specified radix. (There is a second form of digit() that takes a code point. See the following section for a discussion of code points.)

Another method defined by Character is compareTo(), which has the following form:

```java
int compareTo(Character c)
```

It returns zero if the invoking object and c have the same value. It returns a negative value if the invoking object has a lower value. Otherwise, it returns a positive value.
Table 16-7 Various Character Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static boolean isDefined(char ch)</td>
<td>Returns true if ch is defined by Unicode. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isDigit(char ch)</td>
<td>Returns true if ch is a digit. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isIdentifierIgnorable(char ch)</td>
<td>Returns true if ch should be ignored in an identifier. Otherwise, it</td>
</tr>
<tr>
<td></td>
<td>returns false.</td>
</tr>
<tr>
<td>static boolean isISOControl(char ch)</td>
<td>Returns true if ch is an ISO control character. Otherwise, it returns</td>
</tr>
<tr>
<td></td>
<td>false.</td>
</tr>
<tr>
<td>static boolean isJavaIdentifierPart(char ch)</td>
<td>Returns true if ch is allowed as part of a Java identifier (other than the</td>
</tr>
<tr>
<td></td>
<td>first character). Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isJavaIdentifierStart(char ch)</td>
<td>Returns true if ch is allowed as the first character of a Java identifier</td>
</tr>
<tr>
<td></td>
<td>. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isLetter(char ch)</td>
<td>Returns true if ch is a letter. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isLetterOrDigit(char ch)</td>
<td>Returns true if ch is a letter or a digit. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isLowerCase(char ch)</td>
<td>Returns true if ch is a lowercase letter. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isMirrored(char ch)</td>
<td>Returns true if ch is a mirrored Unicode character. A mirrored character</td>
</tr>
<tr>
<td></td>
<td>is one that is reversed for text that is displayed right-to-left.</td>
</tr>
<tr>
<td>static boolean isSpaceChar(char ch)</td>
<td>Returns true if ch is a Unicode space character. Otherwise, it returns</td>
</tr>
<tr>
<td></td>
<td>false.</td>
</tr>
<tr>
<td>static boolean isTitleCase(char ch)</td>
<td>Returns true if ch is a Unicode titlecase character. Otherwise, it returns</td>
</tr>
<tr>
<td></td>
<td>false.</td>
</tr>
<tr>
<td>static boolean isUnicodeIdentifierPart(char ch)</td>
<td>Returns true if ch is allowed as part of a Unicode identifier (other than</td>
</tr>
<tr>
<td></td>
<td>the first character). Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isUnicodeIdentifierStart(char ch)</td>
<td>Returns true if ch is allowed as the first character of a Unicode identifier. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isUpperCase(char ch)</td>
<td>Returns true if ch is an uppercase letter. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static boolean isWhitespace(char ch)</td>
<td>Returns true if ch is whitespace. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static char toLowerCase(char ch)</td>
<td>Returns lowercase equivalent of ch.</td>
</tr>
<tr>
<td>static char toTitleCase(char ch)</td>
<td>Returns titlecase equivalent of ch.</td>
</tr>
<tr>
<td>static char toUpperCase(char ch)</td>
<td>Returns uppercase equivalent of ch.</td>
</tr>
</tbody>
</table>

Character includes a method called `getDirectionality()` which can be used to determine the direction of a character. Several constants are defined that describe directionality. Most programs will not need to use
Character also overrides the equals( ) and hashCode( ) methods.

Two other character-related classes are Character.Subset, used to describe a subset of Unicode, and Character.UnicodeBlock, which contains Unicode character blocks.

### Additions to Character for Unicode Code Point Support

Relatively recently, major additions have been made to Character. Beginning with JDK 5, the Character class has included support for 32-bit Unicode characters. In the past, all Unicode characters could be held by 16 bits, which is the size of a char (and the size of the value encapsulated within a Character), because those values ranged from 0 to FFFF. However, the Unicode character set has been expanded, and more than 16 bits are required. Characters can now range from 0 to 10FFFF.

Here are three important terms. A code point is a character in the range 0 to 10FFFF. Characters that have values greater than FFFF are called supplemental characters. The basic multilingual plane (BMP) are those characters between 0 and FFFF.

The expansion of the Unicode character set caused a fundamental problem for Java. Because a supplemental character has a value greater than a char can hold, some means of handling the supplemental characters was needed. Java addressed this problem two ways. First, Java uses two chars to represent a supplemental character. The first char is called the high surrogate, and
the second is called the low surrogate. New methods, such as codePointAt(), were provided to translate between code points and supplemental characters.

Secondly, Java overloaded several preexisting methods in the Character class. The overloaded forms use int rather than char data. Because an int is large enough to hold any character as a single value, it can be used to store any character. For example, all of the methods in Table 16-7 have overloaded forms that operate on int. Here is a sampling:

```java
static boolean isDigit(int cp)
static boolean isLetter(int cp)
static int toLowerCase(int cp)
```

In addition to the methods overloaded to accept code points, Character adds methods that provide additional support for code points. A sampling is shown in Table 16-8.

**Boolean**

Boolean is a very thin wrapper around boolean values, which is useful mostly when you want to pass a boolean variable by reference. It contains the constants TRUE and FALSE, which define true and false Boolean objects. Boolean also defines the TYPE field, which is the Class object for boolean. Boolean defines these constructors:

```java
Boolean(boolean boolValue)
Boolean(String boolString)
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int charCount(int cp)</td>
<td>Returns 1 if ( cp ) can be represented by a single char. It returns 2 if two chars are needed.</td>
</tr>
<tr>
<td>static int codePointAt(CharSequence chars, int loc)</td>
<td>Returns the code point at the location specified by ( loc ).</td>
</tr>
<tr>
<td>static int codePointAt(char chars[ ], int loc)</td>
<td>Returns the code point at the location specified by ( loc ).</td>
</tr>
<tr>
<td>static int codePointBefore(CharSequence chars, int loc)</td>
<td>Returns the code point at the location that precedes that specified by ( loc ).</td>
</tr>
<tr>
<td>static int codePointBefore(char chars[ ], int loc)</td>
<td>Returns the code point at the location that precedes that specified by ( loc ).</td>
</tr>
<tr>
<td>static boolean isBmpCodePoint(int cp)</td>
<td>Returns true if ( cp ) is part of the basic multilingual plane and false otherwise. (Added by JDK 7.)</td>
</tr>
<tr>
<td>static boolean isHighSurrogate(char ch)</td>
<td>Returns true if ( ch ) contains a valid high surrogate character.</td>
</tr>
<tr>
<td>static boolean isLowSurrogate(char ch)</td>
<td>Returns true if ( ch ) contains a valid low surrogate character.</td>
</tr>
<tr>
<td>static boolean isSupplementaryCodePoint(int cp)</td>
<td>Returns true if ( cp ) contains a supplemental character.</td>
</tr>
<tr>
<td>static boolean isSurrogatePair(char highCh, char lowCh)</td>
<td>Returns true if ( highCh ) and ( lowCh ) form a valid surrogate pair.</td>
</tr>
<tr>
<td>static boolean isValidCodePoint(int cp)</td>
<td>Returns true if ( cp ) contains a valid code point.</td>
</tr>
<tr>
<td>static char[ ] toChars(int cp)</td>
<td>Converts the code point in ( cp ) into its char equivalent, which might require two chars. An array holding the result is returned.</td>
</tr>
<tr>
<td>static int toChars(int cp, char target[ ], int loc)</td>
<td>Converts the code point in ( cp ) into its char equivalent, storing the result in ( target ), beginning at ( loc ). Returns 1 if ( cp ) can be represented by a single char. It returns 2 otherwise.</td>
</tr>
<tr>
<td>static int toCodePoint(char highCh, char lowCh)</td>
<td>Converts ( highCh ) and ( lowCh ) into their equivalent code point.</td>
</tr>
</tbody>
</table>

Table 16-8 A Sampling of Methods That Provide Support for 32-Bit Unicode Code Points

In the first version, boolValue must be either true or false. In the second version, if boolString contains the string “true” (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

Boolean defines the methods shown in Table 16-9.
Table 16-9 The Methods Defined by Boolean

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean booleanValue( )</td>
<td>Returns boolean equivalent.</td>
</tr>
<tr>
<td>static int compare(boolean b1, boolean b2)</td>
<td>Returns zero if b1 and b2 contain the same value. Returns a positive value if b1 is true and b2 is false. Otherwise, returns a negative value. (Added by JDK 7.)</td>
</tr>
<tr>
<td>int compareTo(Boolean b)</td>
<td>Returns zero if the invoking object and b contain the same value. Returns a positive value if the invoking object is true and b is false. Otherwise, returns a negative value.</td>
</tr>
<tr>
<td>boolean equals(Object boolObj)</td>
<td>Returns true if the invoking object is equivalent to boolObj. Otherwise, it returns false.</td>
</tr>
<tr>
<td>static Boolean getBoolean(String propertyName)</td>
<td>Returns true if the system property specified by propertyName is true. Otherwise, it returns false.</td>
</tr>
<tr>
<td>int hashCode( )</td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td>static boolean parseBoolean(String str)</td>
<td>Returns true if str contains the string &quot;true&quot;. Case is not significant. Otherwise, returns false.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns the string equivalent of the invoking object.</td>
</tr>
<tr>
<td>static String toString(boolean boolVal)</td>
<td>Returns the string equivalent of boolVal.</td>
</tr>
<tr>
<td>static Boolean valueOf(boolean boolVal)</td>
<td>Returns the Boolean equivalent of boolVal.</td>
</tr>
<tr>
<td>static Boolean valueOf(String boolString)</td>
<td>Returns true if boolString contains the string &quot;true&quot; (in uppercase or lowercase). Otherwise, it returns false.</td>
</tr>
</tbody>
</table>

Void
The Void class has one field, TYPE, which holds a reference to the Class object for type void. You do not create instances of this class.

Process
The abstract Process class encapsulates a process—that is, an executing program. It is used primarily as a superclass for the type of objects created by exec( ) in the Runtime class, or by start( ) in the ProcessBuilder class. Process contains the abstract methods shown in Table 16-10.
Table 16-10 The Methods Defined by Process (All are abstract.)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void destroy( )</td>
<td>Terminates the process.</td>
</tr>
<tr>
<td>int exitValue( )</td>
<td>Returns an exit code obtained from a subprocess.</td>
</tr>
<tr>
<td>InputStream getErrorStream( )</td>
<td>Returns an input stream that reads input from the process’ err output stream.</td>
</tr>
<tr>
<td>InputStream getInputStream( )</td>
<td>Returns an input stream that reads input from the process’ out output stream.</td>
</tr>
<tr>
<td>OutputStream getOutputStream( )</td>
<td>Returns an output stream that writes output to the process’ in input stream.</td>
</tr>
<tr>
<td>int waitFor( ) throws InterruptedException</td>
<td>Returns the exit code returned by the process. This method does not return until the process on which it is called terminates.</td>
</tr>
</tbody>
</table>

The Runtime class encapsulates the run-time environment. You cannot instantiate a Runtime object. However, you can get a reference to the current Runtime object by calling the static method Runtime.getRuntime(). Once you obtain a reference to the current Runtime object, you can call several methods that control the state and behavior of the Java Virtual Machine. Applets and other untrusted code typically cannot call any of the Runtime methods without raising a SecurityException. Commonly used methods defined by Runtime are shown in Table 16-11.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void addShutdownHook(Thread thrd)</code></td>
<td>Registers <code>thrd</code> as a thread to be run when the Java Virtual Machine terminates.</td>
</tr>
<tr>
<td><code>Process exec(String progName)</code> throws IOException</td>
<td>Executes the program specified by <code>progName</code> as a separate process. An object of type <code>Process</code> is returned that describes the new process.</td>
</tr>
<tr>
<td><code>Process exec(String progName, String environment[])</code> throws IOException</td>
<td>Executes the program specified by <code>progName</code> as a separate process with the environment specified by <code>environment</code>. An object of type <code>Process</code> is returned that describes the new process.</td>
</tr>
<tr>
<td><code>Process exec(String cmdLineArray[])</code> throws IOException</td>
<td>Executes the command line specified by the strings in <code>cmdLineArray</code> as a separate process. An object of type <code>Process</code> is returned that describes the new process.</td>
</tr>
<tr>
<td><code>Process exec(String cmdLineArray[], String environment[])</code> throws IOException</td>
<td>Executes the command line specified by the strings in <code>cmdLineArray</code> as a separate process with the environment specified by <code>environment</code>. An object of type <code>Process</code> is returned that describes the new process.</td>
</tr>
<tr>
<td><code>void exit(int exitCode)</code></td>
<td>Halts execution and returns the value of <code>exitCode</code> to the parent process. By convention, 0 indicates normal termination. All other values indicate some form of error.</td>
</tr>
<tr>
<td><code>long freeMemory()</code></td>
<td>Returns the approximate number of bytes of free memory available to the Java run-time system.</td>
</tr>
<tr>
<td><code>void gc()</code></td>
<td>Initiates garbage collection.</td>
</tr>
<tr>
<td><code>static Runtime getRuntime()</code></td>
<td>Returns the current <code>Runtime</code> object.</td>
</tr>
<tr>
<td><code>void halt(int code)</code></td>
<td>Immediately terminates the Java Virtual Machine. No termination threads or finalizers are run. The value of <code>code</code> is returned to the invoking process.</td>
</tr>
<tr>
<td><code>void load(String libraryFileName)</code></td>
<td>Loads the dynamic library whose file is specified by <code>libraryFileName</code>, which must specify its complete path.</td>
</tr>
<tr>
<td><code>void loadLibrary(String libraryName)</code></td>
<td>Loads the dynamic library whose name is associated with <code>libraryName</code>.</td>
</tr>
<tr>
<td><code>Boolean removeShutdownHook(Thread thrd)</code></td>
<td>Removes <code>thrd</code> from the list of threads to run when the Java Virtual Machine terminates. It returns <code>true</code> if successful—that is, if the thread was removed.</td>
</tr>
<tr>
<td><code>void runFinalization()</code></td>
<td>Initiates calls to the <code>finalize()</code> methods of unused but not yet recycled objects.</td>
</tr>
<tr>
<td><code>long totalMemory()</code></td>
<td>Returns the total number of bytes of memory available to the program.</td>
</tr>
<tr>
<td><code>void traceInstructions(boolean traceOn)</code></td>
<td>Turns on or off instruction tracing, depending upon the value of <code>traceOn</code>. If <code>traceOn</code> is <code>true</code>, the trace is displayed. If it is <code>false</code>, tracing is turned off.</td>
</tr>
<tr>
<td><code>void traceMethodCalls(boolean traceOn)</code></td>
<td>Turns on or off method call tracing, depending upon the value of <code>traceOn</code>. If <code>traceOn</code> is <code>true</code>, the trace is displayed. If it is <code>false</code>, tracing is turned off.</td>
</tr>
</tbody>
</table>
Let’s look at two of the most common uses of the Runtime class: memory management and executing additional processes.

Memory Management

Although Java provides automatic garbage collection, sometimes you will want to know how large the object heap is and how much of it is left. You can use this information, for example, to check your code for efficiency or to approximate how many more objects of a certain type can be instantiated. To obtain these values, use the totalMemory() and freeMemory() methods.

As mentioned in Part I, Java’s garbage collector runs periodically to recycle unused objects. However, sometimes you will want to collect discarded objects prior to the collector’s next appointed rounds. You can run the garbage collector on demand by calling the gc() method. A good thing to try is to call gc() and then call freeMemory() to get a baseline memory usage. Next, execute your code and call freeMemory() again to see how much memory it is allocating. The following program illustrates this idea:

```java
// Demonstrate totalMemory(), freeMemory() and gc().

class MemoryDemo {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        long mem1, mem2;
        Integer someints[] = new Integer[1000];

        System.out.println("Total memory is: " + r.totalMemory());
```

```
```java
mem1 = r.freeMemory();
System.out.println("Initial free memory: " + mem1);
r.gc();
mem1 = r.freeMemory();
System.out.println("Free memory after garbage collection: " + mem1);

for(int i=0; i<1000; i++)
    someints[i] = new Integer(i); // allocate integers

mem2 = r.freeMemory();
System.out.println("Free memory after allocation: " + mem2);
System.out.println("Memory used by allocation: " + (mem1-mem2));

// discard Integers
for(int i=0; i<1000; i++) someints[i] = null;

r.gc(); // request garbage collection
mem2 = r.freeMemory();
System.out.println("Free memory after collecting discarded Integers: " + mem2);
}
```

Sample output from this program is shown here (of course, your actual results may vary):

Total memory is: 1048568
Initial free memory: 751392
Free memory after garbage collection: 841424
Free memory after allocation: 824000
Memory used by allocation: 17424
Free memory after collecting discarded Integers: 842640

Executing Other Programs

In safe environments, you can use Java to execute other heavyweight processes (that is, programs) on your multitasking operating system. Several forms of the exec( ) method allow you to name the program you want to run as well as its input parameters. The exec( ) method
returns a Process object, which can then be used to control how your Java program interacts with this new running process. Because Java can run on a variety of platforms and under a variety of operating systems, exec() is inherently environment-dependent.

The following example uses exec() to launch notepad, Windows’ simple text editor. Obviously, this example must be run under the Windows operating system.

```java
// Demonstrate exec().
class ExecDemo {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        Process p = null;

        try {
            p = r.exec("notepad");
        } catch (Exception e) {
            System.out.println("Error executing notepad.");
        }
    }
}
```

There are several alternative forms of exec(), but the one shown in the example is the most common. The Process object returned by exec() can be manipulated by Process' methods after the new program starts running. You can kill the subprocess with the destroy() method. The waitFor() method causes your program to wait until the subprocess finishes. The exitValue() method returns the value returned by the subprocess when it is finished. This is typically 0 if no problems occur. Here is the preceding exec() example modified to wait for the running process to exit:

```java
// Wait until notepad is terminated.
```
class ExecDemoFini {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        Process p = null;

        try {
            p = r.exec("notepad");
            p.waitFor();
        } catch (Exception e) {
            System.out.println("Error executing notepad.");
        }
        System.out.println("Notepad returned " + p.exitValue());
    }
}

While a subprocess is running, you can write to and read from its standard input and output. The getOutputStream() and getInputStream() methods return the handles to standard in and out of the subprocess. (I/O is examined in detail in Chapter 19.)

ProcessBuilder
ProcessBuilder provides another way to start and manage processes (that is, programs). As explained earlier, all processes are represented by the Process class, and a process can be started by Runtime.exec(). ProcessBuilder offers more control over the processes. For example, you can set the current working directory and change environmental parameters.

ProcessBuilder defines these constructors:

    ProcessBuilder(List<String> args)
    ProcessBuilder(String … args)

Here, args is a list of arguments that specify the name of the program to be executed along with any required command-line arguments. In the first constructor, the
arguments are passed in a List. In the second, they are specified through a varargs parameter. Table 16-12 describes the methods defined by ProcessBuilder.

In Table 16-12, notice that JDK 7 adds several new methods that use the new ProcessBuilder.Redirect class. This abstract class encapsulates an I/O source or target linked to a subprocess. Among other things, these methods enable you to redirect the source or target of I/O operations. For example, you can redirect to a file by calling `to()`, redirect from a file by calling `from()`, and append to a file by calling `appendTo()`. A `File` object linked to the file can be obtained by calling `file()`. These methods are shown here:

```java
static ProcessBuilder.Redirect to(File f)
static ProcessBuilder.Redirect from(File f)
static ProcessBuilder.Redirect appendTo(File f)
File file()
```

Another method supported by ProcessBuilder.Redirect is `type()`, which returns a value of the enumeration type `ProcessBuilder.Redirect.Type`. This enumeration describes the type of the redirection. It defines these values: `APPEND`, `INHERIT`, `PIPE`, `READ`, or `WRITE`. ProcessBuilder.Redirect also defines the constants `INHERIT` and `PIPE`. 
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;String&gt; command()</td>
<td>Returns a reference to a List that contains the name of the program and its arguments. Changes to this list affect the invoking object.</td>
</tr>
<tr>
<td>ProcessBuilder command(List&lt;String&gt; args)</td>
<td>Sets the name of the program and its arguments to those specified by args. Changes to this list affect the invoking object. Returns a reference to the invoking object.</td>
</tr>
<tr>
<td>ProcessBuilder command(String ... args)</td>
<td>Sets the name of the program and its arguments to those specified by args. Returns a reference to the invoking object.</td>
</tr>
<tr>
<td>File directory()</td>
<td>Returns the current working directory of the invoking object. This value will be null if the directory is the same as that of the Java program that started the process.</td>
</tr>
<tr>
<td>ProcessBuilder directory(File dir)</td>
<td>Sets the current working directory of the invoking object. Returns a reference to the invoking object.</td>
</tr>
<tr>
<td>Map&lt;String, String&gt; environment()</td>
<td>Returns the environmental variables associated with the invoking object as key/value pairs.</td>
</tr>
<tr>
<td>ProcessBuilder inheritIO()</td>
<td>Causes the invoked process to use the same source and target for the standard I/O streams as the invoking process. (Added by JDK 7.)</td>
</tr>
<tr>
<td>ProcessBuilder.Redirect redirectError()</td>
<td>Returns the target for standard error as a ProcessBuilder.Redirect object. (Added by JDK 7.)</td>
</tr>
<tr>
<td>ProcessBuilder redirectError(File f)</td>
<td>Sets the target for standard error to the specified file. Returns a reference to the invoking object. (Added by JDK 7.)</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>redirectError(ProcessBuilder.Redirect target)</code></td>
<td>Sets the target for standard error as specified by <code>target</code>. Returns a reference to the invoking object. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>redirectErrorStream()</code></td>
<td>Returns <code>true</code> if the standard error stream has been redirected to the standard output stream. Returns <code>false</code> if the streams are separate.</td>
</tr>
<tr>
<td><code>redirectErrorStream(boolean merge)</code></td>
<td>If <code>merge</code> is <code>true</code>, then the standard error stream is redirected to standard output. If <code>merge</code> is <code>false</code>, the streams are separated, which is the default state. Returns a reference to the invoking object.</td>
</tr>
<tr>
<td><code>redirectInput()</code></td>
<td>Returns the source for standard input as a <code>ProcessBuilder.Redirect</code> object. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>redirectInput(File f)</code></td>
<td>Sets the source for standard input to the specified file. Returns a reference to the invoking object. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>redirectInput(ProcessBuilder.Redirect source)</code></td>
<td>Sets the source for standard input as specified by <code>source</code>. Returns a reference to the invoking object. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>redirectOutput()</code></td>
<td>Returns the target for standard output as a <code>ProcessBuilder.Redirect</code> object. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>redirectOutput(File f)</code></td>
<td>Sets the target for standard output to the specified file. Returns a reference to the invoking object. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>redirectOutput(ProcessBuilder.Redirect target)</code></td>
<td>Sets the target for standard output as specified by <code>target</code>. Returns a reference to the invoking object. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>start()</code> throws IOException</td>
<td>Begins the process specified by the invoking object. In other words, it runs the specified program.</td>
</tr>
</tbody>
</table>

Table 16-12 The Methods Defined by ProcessBuilder

To create a process using ProcessBuilder, simply create an instance of ProcessBuilder, specifying the name of the program and any needed arguments. To begin execution of the program, call `start()` on that instance. Here is an example that executes the Windows text editor `notepad`. Notice that it specifies the name of the file to edit as an argument.

```java
class PBDemo {
    public static void main(String args[]) {
        try {
            ProcessBuilder proc =
```
```java
new ProcessBuilder("notepad.exe", "testfile");
proc.start();
} catch (Exception e) {
    System.out.println("Error executing notepad.");
}
}
```

System

The System class holds a collection of static methods and variables. The standard input, output, and error output of the Java run time are stored in the in, out, and err variables. The methods defined by System are shown in Table 16-13. Many of the methods throw a SecurityException if the operation is not permitted by the security manager.

Let’s look at some common uses of System.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void arraycopy(Object source,</td>
<td>Copies an array. The array to be copied is passed in source, and the index at which point the copy will begin within source is passed in</td>
</tr>
<tr>
<td>int sourceStart, Object target,</td>
<td>sourceStart. The array that will receive the copy is passed in target, and the index at which point the copy will begin within target is passed in</td>
</tr>
<tr>
<td>int targetStart, int size)</td>
<td>targetStart. size is the number of elements that are copied.</td>
</tr>
<tr>
<td>static String clearProperty(String which)</td>
<td>Deletes the environmental variable specified by which. The previous value associated with which is returned.</td>
</tr>
<tr>
<td>static Console console( )</td>
<td>Returns the console associated with the JVM. null is returned if the JVM currently has no console.</td>
</tr>
<tr>
<td>static long currentTimeMillis( )</td>
<td>Returns the current time in terms of milliseconds since midnight, January 1, 1970.</td>
</tr>
<tr>
<td>static void exit(int exitCode)</td>
<td>Halts execution and returns the value of exitCode to the parent process (usually the operating system). By convention, 0 indicates</td>
</tr>
<tr>
<td></td>
<td>normal termination. All other values indicate some form of error.</td>
</tr>
<tr>
<td>static void gc( )</td>
<td>Initiates garbage collection.</td>
</tr>
<tr>
<td>static Map&lt;String, String&gt; getEnv( )</td>
<td>Returns a Map that contains the current environmental variables and their values.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>static String getenv(String which)</td>
<td>Returns the value associated with the environmental variable passed in which.</td>
</tr>
<tr>
<td>static Properties getProperties()</td>
<td>Returns the properties associated with the Java runtime system. (The Properties class is described in Chapter 17.)</td>
</tr>
<tr>
<td>static String getProperty(String which)</td>
<td>Returns the property associated with which. A null object is returned if the desired property is not found.</td>
</tr>
<tr>
<td>static String getProperty(String which, String default)</td>
<td>Returns the property associated with which. If the desired property is not found, default is returned.</td>
</tr>
<tr>
<td>static SecurityManager getSecurityManager()</td>
<td>Returns the current security manager or a null object if no security manager is installed.</td>
</tr>
<tr>
<td>static int identityHashCode(Object obj)</td>
<td>Returns the identity hash code for obj.</td>
</tr>
<tr>
<td>static Channel inheritedChannel() throws IOException</td>
<td>Returns the channel inherited by the Java Virtual Machine. Returns null if no channel is inherited.</td>
</tr>
<tr>
<td>static String lineSeparator()</td>
<td>Returns a string that contains the line-separator characters.</td>
</tr>
<tr>
<td>static void load(String libraryFileName)</td>
<td>Loads the dynamic library whose file is specified by libraryFileName, which must specify its complete path.</td>
</tr>
<tr>
<td>static void loadLibrary(String libraryName)</td>
<td>Loads the dynamic library whose name is associated with libraryName.</td>
</tr>
<tr>
<td>static String mapLibraryName(String lib)</td>
<td>Returns a platform-specific name for the library named lib.</td>
</tr>
<tr>
<td>static long nanoTime()</td>
<td>Obtains the most precise timer in the system and returns its value in terms of nanoseconds since some arbitrary starting point. The accuracy of the timer is unknowable.</td>
</tr>
<tr>
<td>static void runFinalization()</td>
<td>Initiates calls to the finalize() methods of unused but not yet recycled objects.</td>
</tr>
<tr>
<td>static void setErr(PrintStream eStream)</td>
<td>Sets the standard err stream to eStream.</td>
</tr>
<tr>
<td>static void setIn(InputStream iStream)</td>
<td>Sets the standard in stream to iStream.</td>
</tr>
<tr>
<td>static void setOut(PrintStream oStream)</td>
<td>Sets the standard out stream to oStream.</td>
</tr>
<tr>
<td>static void setProperties(Properties sysProperties)</td>
<td>Sets the current system properties as specified by sysProperties.</td>
</tr>
<tr>
<td>static String setProperty(String which, String v)</td>
<td>Assigns the value v to the property named which.</td>
</tr>
<tr>
<td>static void setSecurityManager(SecurityManager secMan)</td>
<td>Sets the security manager to that specified by secMan.</td>
</tr>
</tbody>
</table>

Table 16-13 The Methods Defined by System

Using currentTimeMillis( ) to Time Program Execution

One use of the System class that you might find particularly interesting is to use the currentTimeMillis( )
method to time how long various parts of your program take to execute. The `currentTimeMillis()` method returns the current time in terms of milliseconds since midnight, January 1, 1970. To time a section of your program, store this value just before beginning the section in question. Immediately upon completion, call `currentTimeMillis()` again. The elapsed time will be the ending time minus the starting time. The following program demonstrates this:

```java
// Timing program execution.

class Elapsed {
    public static void main(String args[]) {
        long start, end;

        System.out.println("Timing a for loop from 0 to 100,000,000");
        // time a for loop from 0 to 100,000,000
        start = System.currentTimeMillis(); // get starting time
        for(long i=0; i < 100000000L; i++) ;
        end = System.currentTimeMillis(); // get ending time

        System.out.println("Elapsed time: " + (end-start));
    }
}
```

Here is a sample run (remember that your results probably will differ):

```
Timing a for loop from 0 to 100,000,000
Elapsed time: 10
```

If your system has a timer that offers nanosecond precision, then you could rewrite the preceding program to use `nanoTime()` rather than `currentTimeMillis()`. For example, here is the key portion of the program rewritten to use `nanoTime()`:

```java
// Timing program execution.

class Elapsed {
    public static void main(String args[]) {
        long start, end;

        System.out.println("Timing a for loop from 0 to 100,000,000");
        // time a for loop from 0 to 100,000,000
        start = System.currentTimeMillis(); // get starting time
        for(long i=0; i < 100000000L; i++) ;
        end = System.currentTimeMillis(); // get ending time

        System.out.println("Elapsed time: " + (end-start));
    }
}
```
Using arraycopy( )

The arraycopy( ) method can be used to copy quickly an array of any type from one place to another. This is much faster than the equivalent loop written out longhand in Java. Here is an example of two arrays being copied by the arraycopy( ) method. First, a is copied to b. Next, all of a’s elements are shifted down by one. Then, b is shifted up by one.

```
// Using arraycopy().

class ACDemo {
    static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };  
    static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };  

    public static void main(String args[]) {
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, b, 0, a.length);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, a, 1, a.length - 1);
        System.arraycopy(b, 1, b, 0, b.length - 1);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
    }
}
```

As you can see from the following output, you can copy using the same source and destination in either direction:

```
a = ABCDEFGHIJ  
b = MMMMMMMMMM  
a = ABCDEFGHIJ  
b = ABCDEFGHIJ  
a = AABCDEFGHIJ  
a = BCDEFGHIJJ  
```
Environment Properties

The following properties are available in all cases:

<table>
<thead>
<tr>
<th>Property</th>
<th>Property</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>file.separator</td>
<td>java.specification.version</td>
<td>java.vm.version</td>
</tr>
<tr>
<td>java.class.path</td>
<td>java.vendor</td>
<td>line.separator</td>
</tr>
<tr>
<td>java.class.version</td>
<td>java.vendor.url</td>
<td>os.arch</td>
</tr>
<tr>
<td>java.compiler</td>
<td>java.version</td>
<td>os.name</td>
</tr>
<tr>
<td>java.ext.dirs</td>
<td>java.vm.name</td>
<td>os.version</td>
</tr>
<tr>
<td>java.home</td>
<td>java.vm.specification.name</td>
<td>path.separator</td>
</tr>
<tr>
<td>java.io.tmpdir</td>
<td>java.vm.specification.vendor</td>
<td>user.dir</td>
</tr>
<tr>
<td>java.library.path</td>
<td>java.library.path</td>
<td>user.home</td>
</tr>
<tr>
<td>java.specification.name</td>
<td>java.vm.vendor</td>
<td>user.name</td>
</tr>
<tr>
<td>java.specification.vendor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can obtain the values of various environment variables by calling the System.getProperty() method. For example, the following program displays the path to the current user directory:

```java
class ShowUserDir {
    public static void main(String args[]) {
        System.out.println(System.getProperty("user.dir"));
    }
}
```

Object

As mentioned in Part I, Object is a superclass of all other classes. Object defines the methods shown in Table 16-14, which are available to every object.

Using clone() and the Cloneable Interface

Most of the methods defined by Object are discussed elsewhere in this book. However, one deserves special attention: clone(). The clone() method generates a duplicate copy of the object on which it is called. Only classes that implement the Cloneable interface can be cloned.
The Cloneable interface defines no members. It is used to indicate that a class allows a bitwise copy of an object (that is, a clone) to be made. If you try to call clone( ) on a class that does not implement Cloneable, a CloneNotSupportedException is thrown. When a clone is made, the constructor for the object being cloned is not called. A clone is simply an exact copy of the original.

Cloning is a potentially dangerous action, because it can cause unintended side effects. For example, if the object being cloned contains a reference variable called obRef, then when the clone is made, obRef in the clone will refer to the same object as does obRef in the original. If the clone makes a change to the contents of the object referred to by obRef, then it will be changed for the original object, too. Here is another example: If an object opens an I/O stream and is then cloned, two objects will be capable of operating on the same stream. Further, if one of these objects closes the stream, the other object might still attempt to write to it, causing an error. In some cases, you will need to override the clone( ) method defined by Object to handle these types of problems.
Because cloning can cause problems, clone( ) is declared as protected inside Object. This means that it must either be called from within a method defined by the class that implements Cloneable, or it must be explicitly overridden by that class so that it is public. Let’s look at an example of each approach.

The following program implements Cloneable and defines the method cloneTest( ), which calls clone( ) in Object:

```java
// Demonstrate the clone() method

class TestClone implements Cloneable {
    int a;
    double b;

    // This method calls Object's clone().
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object clone( )</td>
<td>Creates a new object that is the same as the invoking object.</td>
</tr>
<tr>
<td>throws CloneNotSupportedException</td>
<td></td>
</tr>
<tr>
<td>boolean equals(Object object)</td>
<td>Returns true if the invoking object is equivalent to object.</td>
</tr>
<tr>
<td>void finalize( ) throws Throwable</td>
<td>Default finalize() method. It is called before an unused object is recycled.</td>
</tr>
<tr>
<td>final Class&lt;?&gt; getClass( )</td>
<td>Obtains a Class object that describes the invoking object.</td>
</tr>
<tr>
<td>int hashCode( )</td>
<td>Returns the hash code associated with the invoking object.</td>
</tr>
<tr>
<td>final void notify( )</td>
<td>Resumes execution of a thread waiting on the invoking object.</td>
</tr>
<tr>
<td>final void notifyAll( )</td>
<td>Resumes execution of all threads waiting on the invoking object.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns a string that describes the object.</td>
</tr>
<tr>
<td>final void wait( )</td>
<td>Waits on another thread of execution.</td>
</tr>
<tr>
<td>throws InterruptedException</td>
<td></td>
</tr>
<tr>
<td>final void wait(long milliseconds) throws InterruptedException</td>
<td>Waits up to the specified number of milliseconds on another thread of execution.</td>
</tr>
<tr>
<td>final void wait(long milliseconds, int nanoseconds) throws InterruptedException</td>
<td>Waits up to the specified number of milliseconds plus nanoseconds on another thread of execution.</td>
</tr>
</tbody>
</table>
TestClone cloneTest() {
    try {
        // call clone in Object.
        return (TestClone) super.clone();
    } catch(CloneNotSupportedException e) {
        System.out.println("Cloning not allowed.");
        return this;
    }
}
}

class CloneDemo {
    public static void main(String args[]) {
        TestClone x1 = new TestClone();
        TestClone x2;

        x1.a = 10;
        x1.b = 20.98;

        x2 = x1.cloneTest(); // clone x1

        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
}

Here, the method cloneTest() calls clone() in Object and returns the result. Notice that the object returned by clone() must be cast into its appropriate type (TestClone).

The following example overrides clone() so that it can be called from code outside of its class. To do this, its access specifier must be public, as shown here:

    // Override the clone() method.

    class TestClone implements Cloneable {
        int a;
        double b;

        // clone() is now overridden and is public.
        public Object clone() {
            try {
                // call clone in Object.
                return super.clone();
            } catch(CloneNotSupportedException e) {
                System.out.println("Cloning not allowed.");
                return this;
            }
        }
    }
catch(CloneNotSupportedException e) {
    System.out.println("Cloning not allowed.");
    return this;
}
}

class CloneDemo2 {
    public static void main(String args[]) {
        TestClone x1 = new TestClone();
        TestClone x2;

        x1.a = 10;
        x1.b = 20.98;

        // here, clone() is called directly.
        x2 = (TestClone) x1.clone();

        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
}

The side effects caused by cloning are sometimes difficult to see at first. It is easy to think that a class is safe for cloning when it actually is not. In general, you should not implement Cloneable for any class without good reason.

Class
Class encapsulates the run-time state of a class or interface. Objects of type Class are created automatically, when classes are loaded. You cannot explicitly declare a Class object. Generally, you obtain a Class object by calling the getClass( ) method defined by Object. Class is a generic type that is declared as shown here:

    class Class<T>

Here, T is the type of the class or interface represented.
A sampling of commonly used methods defined by Class is shown in Table 16-15.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Class&lt;?&gt; forName(String name) throws ClassNotFoundException</td>
<td>Returns a Class object given its complete name.</td>
</tr>
<tr>
<td>static Class&lt;?&gt; forName(String name, boolean how, ClassLoader ldr)</td>
<td>Returns a Class object given its complete name. The object is loaded using</td>
</tr>
<tr>
<td></td>
<td>the loader specified by ldr. If how is true, the object is initialized;</td>
</tr>
<tr>
<td></td>
<td>otherwise, it is not.</td>
</tr>
<tr>
<td>&lt;A extends Annotation&gt; A</td>
<td>Returns an Annotation object that contains the annotation associated with</td>
</tr>
<tr>
<td>getAnnotation(Class&lt;A&gt; annoType)</td>
<td>annoType for the invoking object.</td>
</tr>
<tr>
<td>Annotation[ ] getAnnotations( )</td>
<td>Obtains all annotations associated with the invoking object and stores</td>
</tr>
<tr>
<td></td>
<td>them in an array of Annotation objects. Returns a reference to this array.</td>
</tr>
<tr>
<td>Class&lt;?&gt;[ ] getClasses( )</td>
<td>Returns a Class object for each public class and interface that is a member</td>
</tr>
<tr>
<td></td>
<td>of the class represented by the invoking object.</td>
</tr>
<tr>
<td>ClassLoader getClassLoader( )</td>
<td>Returns the ClassLoader object that loaded the class or interface.</td>
</tr>
<tr>
<td>Constructor&lt;T&gt; getConstructor(Class&lt;?&gt; ... paramTypes) throws</td>
<td>Returns a Constructor object that represents the constructor for the class</td>
</tr>
<tr>
<td>NoSuchMethodException, SecurityException</td>
<td>represented by the invoking object that has the parameter types specified</td>
</tr>
<tr>
<td></td>
<td>by paramTypes.</td>
</tr>
<tr>
<td>Constructor&lt;?&gt;[ ] getConstructors( ) throws SecurityException</td>
<td>Obtains a Constructor object for each public constructor of the class</td>
</tr>
<tr>
<td></td>
<td>represented by the invoking object and stores them in an array. Returns a</td>
</tr>
<tr>
<td></td>
<td>reference to this array.</td>
</tr>
<tr>
<td>Annotation[ ] getDeclaredAnnotations( )</td>
<td>Obtains an Annotation object for all the annotations that are declared by</td>
</tr>
<tr>
<td></td>
<td>the invoking object and stores them in an array. Returns a reference to</td>
</tr>
<tr>
<td></td>
<td>this array. (Inherited annotations are ignored.)</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Constructor&lt; ? &gt;[ ] getDeclaredConstructors() throws SecurityException</td>
<td>Obtains a <strong>Constructor</strong> object for each constructor declared by the class represented by the invoking object and stores them in an array. Returns a reference to this array. (Superclass constructors are ignored.)</td>
</tr>
<tr>
<td>Field[ ] getDeclaredFields() throws SecurityException</td>
<td>Obtains a <strong>Field</strong> object for each field declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited fields are ignored.)</td>
</tr>
<tr>
<td>Method[ ] getDeclaredMethods() throws SecurityException</td>
<td>Obtains a <strong>Method</strong> object for each method declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited methods are ignored.)</td>
</tr>
<tr>
<td>Field getField(String fieldName) throws NoSuchMethodException, SecurityException</td>
<td>Returns a <strong>Field</strong> object that represents the public field specified by <code>fieldName</code> for the class or interface represented by the invoking object.</td>
</tr>
<tr>
<td>Field[ ] getFields() throws SecurityException</td>
<td>Obtains a <strong>Field</strong> object for each public field of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.</td>
</tr>
<tr>
<td>Class&lt; ? &gt;[ ] getInterfaces()</td>
<td>When invoked on an object that represents a class, this method returns an array of the interfaces implemented by that class. When invoked on an object that represents an interface, this method returns an array of interfaces extended by that interface.</td>
</tr>
</tbody>
</table>
Table 16-15 A Sampling of Methods Defined by Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object.getMethod(String methName, Class&lt;?&gt; ... paramTypes) throws NoSuchMethodException, SecurityException</td>
<td>Returns a Method object that represents the public method specified by methName and having the parameter types specified by paramTypes in the class or interface represented by the invoking object.</td>
</tr>
<tr>
<td>Method[] getMethods() throws SecurityException</td>
<td>Obtains a Method object for each public method of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.</td>
</tr>
<tr>
<td>String getName()</td>
<td>Returns the complete name of the class or interface of the type represented by the invoking object.</td>
</tr>
<tr>
<td>ProtectionDomain getProtectionDomain()</td>
<td>Returns the protection domain associated with the invoking object.</td>
</tr>
<tr>
<td>Class&lt;? super T&gt; getSuperclass()</td>
<td>Returns the superclass of the type represented by the invoking object. The return value is null if the represented type is Object or not a class.</td>
</tr>
<tr>
<td>boolean isInterface()</td>
<td>Returns true if the type represented by the invoking object is an interface. Otherwise, it returns false.</td>
</tr>
<tr>
<td>T newInstance() throws IllegalAccessException, InstantiationException</td>
<td>Creates a new instance (i.e., a new object) that is of the same type as that represented by invoking object. This is equivalent to using new with the class’ default constructor. The new object is returned. This method will fail if the represented type is abstract, not a class, or does not have a default constructor.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns the string representation of the type represented by the invoking object or interface.</td>
</tr>
</tbody>
</table>

The methods defined by Class are often useful in situations where run-time type information about an object is required. As Table 16-15 shows, methods are provided that allow you to determine additional information about a particular class, such as its public constructors, fields, and methods. Among other things, this is important for the Java Beans functionality, which is discussed later in this book.

The following program demonstrates getClass( ) (inherited from Object) and getSuperclass( ) (from Class):
// Demonstrate Run-Time Type Information.

class X {
    int a;
    float b;
}

class Y extends X {
    double c;
}

class RTTI {
    public static void main(String args[]) {
        X x = new X();
        Y y = new Y();
        Class<?> c1Obj;

        c1Obj = x.getClass(); // get Class reference
        System.out.println("x is object of type: " +
                            c1Obj.getName());

        c1Obj = y.getClass(); // get Class reference
        System.out.println("y is object of type: " +
                            c1Obj.getName());

        c1Obj = c1Obj.getSuperclass();
        System.out.println("y's superclass is " +
                            c1Obj.getName());
    }
}

The output from this program is shown here:

    x is object of type: X
    y is object of type: Y
    y's superclass is X

ClassLoader

The abstract class ClassLoader defines how classes are loaded. Your application can create subclasses that extend ClassLoader, implementing its methods. Doing so allows you to load classes in some way other than the way they are normally loaded by the Java run-time system. However, this is not something that you will normally need to do.
The Math class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods. Math defines two double constants: E (approximately 2.72) and PI (approximately 3.14).

Trigonometric Functions

The following methods accept a double parameter for an angle in radians and return the result of their respective trigonometric function:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static double sin(double arg)</td>
<td>Returns the sine of the angle specified by arg in radians.</td>
</tr>
<tr>
<td>static double cos(double arg)</td>
<td>Returns the cosine of the angle specified by arg in radians.</td>
</tr>
<tr>
<td>static double tan(double arg)</td>
<td>Returns the tangent of the angle specified by arg in radians.</td>
</tr>
</tbody>
</table>

The next methods take as a parameter the result of a trigonometric function and return, in radians, the angle that would produce that result. They are the inverse of their non-arc companions:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static double asin(double arg)</td>
<td>Returns the angle whose sine is specified by arg.</td>
</tr>
<tr>
<td>static double acos(double arg)</td>
<td>Returns the angle whose cosine is specified by arg.</td>
</tr>
<tr>
<td>static double atan(double arg)</td>
<td>Returns the angle whose tangent is specified by arg.</td>
</tr>
<tr>
<td>static double atan2(double x, double y)</td>
<td>Returns the angle whose tangent is x/y.</td>
</tr>
</tbody>
</table>

The next methods compute the hyperbolic sine, cosine, and tangent of an angle:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static double sinh(double arg)</td>
<td>Returns the hyperbolic sine of the angle specified by arg.</td>
</tr>
<tr>
<td>static double cosh(double arg)</td>
<td>Returns the hyperbolic cosine of the angle specified by arg.</td>
</tr>
<tr>
<td>static double tanh(double arg)</td>
<td>Returns the hyperbolic tangent of the angle specified by arg.</td>
</tr>
</tbody>
</table>
Math defines the following exponential methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static double cbrt(double arg)</td>
<td>Returns the cube root of arg.</td>
</tr>
<tr>
<td>static double exp(double arg)</td>
<td>Returns e to the arg.</td>
</tr>
<tr>
<td>static double expm1 (double arg)</td>
<td>Returns e to the arg - 1.</td>
</tr>
<tr>
<td>static double log(double arg)</td>
<td>Returns the natural logarithm of arg.</td>
</tr>
<tr>
<td>static double log10(double arg)</td>
<td>Returns the base 10 logarithm for arg.</td>
</tr>
<tr>
<td>static double log1p(double arg)</td>
<td>Returns the natural logarithm for arg + 1.</td>
</tr>
<tr>
<td>static double pow(double y, double x)</td>
<td>Returns y raised to the x; for example, pow(2.0, 3.0) returns 8.0.</td>
</tr>
<tr>
<td>static double scalb(double arg, int factor)</td>
<td>Returns arg × 2^{factor}.</td>
</tr>
<tr>
<td>static float scalb(float arg, int factor)</td>
<td>Returns arg × 2^{factor}.</td>
</tr>
<tr>
<td>static double sqrt(double arg)</td>
<td>Returns the square root of arg.</td>
</tr>
</tbody>
</table>

Rounding Functions

The Math class defines several methods that provide various types of rounding operations. They are shown in Table 16-16. Notice the two ulp( ) methods at the end of the table. In this context, ulp stands for units in the last place. It indicates the distance between a value and the next higher value. It can be used to help assess the accuracy of a result.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int abs(int arg)</td>
<td>Returns the absolute value of arg.</td>
</tr>
<tr>
<td>static long abs(long arg)</td>
<td>Returns the absolute value of arg.</td>
</tr>
<tr>
<td>static float abs(float arg)</td>
<td>Returns the absolute value of arg.</td>
</tr>
<tr>
<td>static double abs(double arg)</td>
<td>Returns the absolute value of arg.</td>
</tr>
<tr>
<td>static double ceil(double arg)</td>
<td>Returns the smallest whole number greater than or equal to arg.</td>
</tr>
<tr>
<td>static double floor(double arg)</td>
<td>Returns the largest whole number less than or equal to arg.</td>
</tr>
<tr>
<td>static int max(int x, int y)</td>
<td>Returns the maximum of x and y.</td>
</tr>
<tr>
<td>static long max(long x, long y)</td>
<td>Returns the maximum of x and y.</td>
</tr>
<tr>
<td>static float max(float x, float y)</td>
<td>Returns the maximum of x and y.</td>
</tr>
<tr>
<td>static double max(double x, double y)</td>
<td>Returns the maximum of x and y.</td>
</tr>
<tr>
<td>static int min(int x, int y)</td>
<td>Returns the minimum of x and y.</td>
</tr>
<tr>
<td>static long min(long x, long y)</td>
<td>Returns the minimum of x and y.</td>
</tr>
<tr>
<td>static float min(float x, float y)</td>
<td>Returns the minimum of x and y.</td>
</tr>
<tr>
<td>static double min(double x, double y)</td>
<td>Returns the minimum of x and y.</td>
</tr>
<tr>
<td>static double nextAfter(double arg, double toward)</td>
<td>Beginning with the value of arg, returns the next value in the direction of toward. If ( \text{arg} = \text{toward} ), then ( \text{toward} ) is returned.</td>
</tr>
<tr>
<td>static float nextAfter(float arg, double toward)</td>
<td>Beginning with the value of arg, returns the next value in the direction of toward. If ( \text{arg} = \text{toward} ), then ( \text{toward} ) is returned.</td>
</tr>
<tr>
<td>static double nextUp(double arg)</td>
<td>Returns the next value in the positive direction from ( \text{arg} ).</td>
</tr>
<tr>
<td>static float nextUp(float arg)</td>
<td>Returns the next value in the positive direction from ( \text{arg} ).</td>
</tr>
<tr>
<td>static double rint(double arg)</td>
<td>Returns the integer nearest in value to ( \text{arg} ).</td>
</tr>
<tr>
<td>static int round(float arg)</td>
<td>Returns ( \text{arg} ) rounded up to the nearest int.</td>
</tr>
<tr>
<td>static long round(double arg)</td>
<td>Returns ( \text{arg} ) rounded up to the nearest long.</td>
</tr>
<tr>
<td>static float ulp(float arg)</td>
<td>Returns the ulp for ( \text{arg} ).</td>
</tr>
<tr>
<td>static double ulp(double arg)</td>
<td>Returns the ulp for ( \text{arg} ).</td>
</tr>
</tbody>
</table>

Table 16-16 The Rounding Methods Defined by Math

Miscellaneous Math Methods

In addition to the methods just shown, Math defines several other methods, which are shown here:
The following program demonstrates `toRadians()` and `toDegrees()`:

```java
// Demonstrate toDegrees() and toRadians().
class Angles {
    public static void main(String args[]) {
        double theta = 120.0;

        System.out.println(theta + " degrees is " +
                           Math.toRadians(theta) + " radians.");

        theta = 1.312;
        System.out.println(theta + " radians is " +
                           Math.toDegrees(theta) + " degrees.");
    }
}
```

The output is shown here:

120.0 degrees is 2.0943951023931953 radians.
1.312 radians is 75.17206272116401 degrees.
StrictMath

The StrictMath class defines a complete set of mathematical methods that parallel those in Math. The difference is that the StrictMath version is guaranteed to generate precisely identical results across all Java implementations, whereas the methods in Math are given more latitude in order to improve performance.

Compiler

The Compiler class supports the creation of Java environments in which Java bytecode is compiled into executable code rather than interpreted. It is not for normal programming use.

Thread, ThreadGroup, and Runnable

The Runnable interface and the Thread and ThreadGroup classes support multithreaded programming. Each is examined next.

NOTE

An overview of the techniques used to manage threads, implement the Runnable interface, and create multithreaded programs is presented in Chapter 11.

The Runnable Interface

The Runnable interface must be implemented by any class that will initiate a separate thread of execution. Runnable only defines one abstract method, called run(), which is the entry point to the thread. It is defined like this:
void run( )

Threads that you create must implement this method.

Thread

Thread creates a new thread of execution. It implements Runnable and defines the following commonly used constructors:

   Thread( )
   Thread(Runnable threadOb)
   Thread(Runnable threadOb, String threadName)
   Thread(String threadName)
   Thread(ThreadGroup groupOb, Runnable threadOb)
   Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)
   Thread(ThreadGroup groupOb, String threadName)

threadOb is an instance of a class that implements the Runnable interface and defines where execution of the thread will begin. The name of the thread is specified by threadName. When a name is not specified, one is created by the Java Virtual Machine. groupOb specifies the thread group to which the new thread will belong. When no thread group is specified, the new thread belongs to the same group as the parent thread.

   The following constants are defined by Thread:

       MAX_PRIORITY
       MIN_PRIORITY
       NORM_PRIORITY
As expected, these constants specify the maximum, minimum, and default thread priorities.

The methods defined by Thread are shown in Table 16-17. In early versions of Java, Thread also included the methods stop( ), suspend( ), and resume( ). However, as explained in Chapter 11, these were deprecated because they were inherently unstable. Also deprecated are countStackFrames( ), because it calls suspend( ), and destroy( ), because it can cause deadlock.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int activeCount( )</td>
<td>Returns the approximate number of active threads in the group to which the thread belongs.</td>
</tr>
<tr>
<td>final void checkAccess( )</td>
<td>Causes the security manager to verify that the current thread can access and/or change the thread on which checkAccess( ) is called.</td>
</tr>
<tr>
<td>static Thread currentThread( )</td>
<td>Returns a Thread object that encapsulates the thread that calls this method.</td>
</tr>
<tr>
<td>static void dumpStack( )</td>
<td>Displays the call stack for the thread.</td>
</tr>
<tr>
<td>static int enumerate(Thread threads[ ])</td>
<td>Puts copies of all Thread objects in the current thread’s group into threads. The number of threads is returned.</td>
</tr>
<tr>
<td>static Map&lt;Thread, StackTraceElement[ ]&gt;</td>
<td>Returns a Map that contains the stack traces for all active threads. In the map, each entry consists of a key, which is the Thread object, and its value, which is an array of StackTraceElement.</td>
</tr>
<tr>
<td>ClassLoader getContextClassLoader( )</td>
<td>Returns the context class loader that is used to load classes and resources for this thread.</td>
</tr>
<tr>
<td>static Thread.UncaughtExceptionHandler</td>
<td>Returns the default uncaught exception handler.</td>
</tr>
<tr>
<td>getDefaultUncaughtExceptionHandler( )</td>
<td></td>
</tr>
<tr>
<td>long getID( )</td>
<td>Returns the ID of the invoking thread.</td>
</tr>
<tr>
<td>final String getName( )</td>
<td>Returns the thread’s name.</td>
</tr>
<tr>
<td>final int getPriority( )</td>
<td>Returns the thread’s priority setting.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>StackTraceElement[ ] getStackTrace()</td>
<td>Returns an array containing the stack trace for the invoking thread.</td>
</tr>
<tr>
<td>Thread.State getState()</td>
<td>Returns the invoking thread’s state.</td>
</tr>
<tr>
<td>final ThreadGroup getThreadGroup()</td>
<td>Returns the ThreadGroup object of which the invoking thread is a member.</td>
</tr>
<tr>
<td>Thread.UncaughtExceptionHandler</td>
<td>Returns the invoking thread’s uncaught exception handler.</td>
</tr>
<tr>
<td>static boolean holdsLock(Object ob)</td>
<td>Returns true if the invoking thread owns the lock on ob. Returns false other-</td>
</tr>
<tr>
<td>static void interrupt()</td>
<td>Checks if the thread is interrupted.</td>
</tr>
<tr>
<td>static boolean interrupted()</td>
<td>Returns true if the currently executing thread has been scheduled for inter-</td>
</tr>
<tr>
<td>final boolean isAlive()</td>
<td>Returns true if the thread is still active. Otherwise, it returns false.</td>
</tr>
<tr>
<td>final boolean isDaemon()</td>
<td>Returns true if the thread is a daemon thread. Otherwise, it returns false.</td>
</tr>
<tr>
<td>boolean isInterrupted()</td>
<td>Returns true if the thread is interrupted. Otherwise, it returns false.</td>
</tr>
<tr>
<td>void join()</td>
<td>Waits until the thread terminates.</td>
</tr>
<tr>
<td>throws InterruptedException</td>
<td></td>
</tr>
<tr>
<td>final void join(long milliseconds)</td>
<td>Waits up to the specified number of milliseconds for the thread on which it</td>
</tr>
<tr>
<td>throws InterruptedException</td>
<td>is called to terminate.</td>
</tr>
<tr>
<td>final void join(long milliseconds, int nano-</td>
<td>Waits up to the specified number of milliseconds plus nanoseconds for the</td>
</tr>
<tr>
<td>microseconds, int nanoseconds)</td>
<td>thread on which it is called to terminate.</td>
</tr>
<tr>
<td>throws InterruptedException</td>
<td></td>
</tr>
<tr>
<td>void run()</td>
<td>Begins execution of a thread.</td>
</tr>
<tr>
<td>void setContextClassLoader(ClassLoader cl)</td>
<td>Sets the context class loader that will be used by the invoking thread to cl.</td>
</tr>
<tr>
<td>final void setDaemon(boolean state)</td>
<td>Flags the thread as a daemon thread.</td>
</tr>
<tr>
<td>static void</td>
<td>Sets the default uncaught exception handler to e.</td>
</tr>
<tr>
<td>setDefaultUncaughtExceptionHandler(</td>
<td></td>
</tr>
<tr>
<td>Thread.UncaughtExceptionHandler e)</td>
<td></td>
</tr>
<tr>
<td>final void setName(String threadName)</td>
<td>Sets the name of the thread to that specified by threadName.</td>
</tr>
<tr>
<td>final void setPriority(int priority)</td>
<td>Sets the priority of the thread to that specified by priority.</td>
</tr>
</tbody>
</table>
ThreadGroup

ThreadGroup creates a group of threads. It defines these two constructors:

```
ThreadGroup(String groupName)
ThreadGroup(ThreadGroup parentOb, String groupName)
```

For both forms, groupName specifies the name of the thread group. The first version creates a new group that has the current thread as its parent. In the second form, the parent is specified by parentOb. The non-deprecated methods defined by ThreadGroup are shown in Table 16-18.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int activeCount( )</td>
<td>Returns the approximate number of active threads in the invoking group (including those in subgroups).</td>
</tr>
<tr>
<td>int activeGroupCount( )</td>
<td>Returns the approximate number of active groups (including subgroups) for which the invoking thread is a parent.</td>
</tr>
<tr>
<td>final void checkAccess( )</td>
<td>Causes the security manager to verify that the invoking thread may access and/or change the group on which checkAccess( ) is called.</td>
</tr>
<tr>
<td>final void destroy( )</td>
<td>Destroys the thread group (and any child groups) on which it is called.</td>
</tr>
<tr>
<td>int enumerate(Thread group[ ])</td>
<td>Puts the active threads that comprise the invoking thread group (including those in subgroups) into the group array.</td>
</tr>
<tr>
<td>int enumerate(Thread group[ ], boolean all)</td>
<td>Puts the active threads that comprise the invoking thread group into the group array. If all is true, then threads in all subgroups of the thread are also put into group.</td>
</tr>
<tr>
<td>int enumerate(ThreadGroup group[ ])</td>
<td>Puts the active subgroups (including subgroups of subgroups and so on) of the invoking thread group into the group array.</td>
</tr>
<tr>
<td>int enumerate(ThreadGroup group[ ], boolean all)</td>
<td>Puts the active subgroups of the invoking thread group into the group array. If all is true, then all active subgroups of the subgroups (and so on) are also put into group.</td>
</tr>
<tr>
<td>final int getMaxPriority( )</td>
<td>Returns the maximum priority setting for the group.</td>
</tr>
<tr>
<td>final String getName( )</td>
<td>Returns the name of the group.</td>
</tr>
<tr>
<td>final ThreadGroup getParent( )</td>
<td>Returns null if the invoking ThreadGroup object has no parent. Otherwise, it returns the parent of the invoking object.</td>
</tr>
<tr>
<td>final void interrupt( )</td>
<td>Invokes the interrupt( ) method of all threads in the group and any subgroups.</td>
</tr>
<tr>
<td>final boolean isDaemon( )</td>
<td>Returns true if the group is a daemon group. Otherwise, it returns false.</td>
</tr>
<tr>
<td>boolean isDestroyed( )</td>
<td>Returns true if the group has been destroyed. Otherwise, it returns false.</td>
</tr>
<tr>
<td>void list( )</td>
<td>Displays information about the group.</td>
</tr>
<tr>
<td>final boolean parentOf(ThreadGroup group)</td>
<td>Returns true if the invoking thread is the parent of group (or group, itself). Otherwise, it returns false.</td>
</tr>
<tr>
<td>final void setDaemon(boolean isDaemon)</td>
<td>If isDaemon is true, then the invoking group is flagged as a daemon group.</td>
</tr>
<tr>
<td>final void setMaxPriority(int priority)</td>
<td>Sets the maximum priority of the invoking group to priority.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns the string equivalent of the group.</td>
</tr>
<tr>
<td>void uncaughtException(Thread thread, Throwable e)</td>
<td>This method is called when an exception goes uncaught.</td>
</tr>
</tbody>
</table>

Table 16-18 The Methods Defined by ThreadGroup
Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads. For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience. The following program, which creates two thread groups of two threads each, illustrates this usage:
Demonstrate thread groups.

class NewThread extends Thread {
    boolean suspendFlag;

    NewThread(String threadname, ThreadGroup tgOb) {
        super(tgOb, threadname);
        System.out.println("New thread: " + this);
        suspendFlag = false;
        start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(getName() + ": " + i);
                Thread.sleep(1000);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (Exception e) {
            System.out.println("Exception in " + getName());
        }
        System.out.println(getName() + " exiting.");
    }
}
synchronized void mysuspend() {
    suspendFlag = true;
}
synchronized void myresume() {
    suspendFlag = false;
    notify();
}

class ThreadGroupDemo {
    public static void main(String[] args) {
        ThreadGroup groupA = new ThreadGroup("Group A");
        ThreadGroup groupB = new ThreadGroup("Group B");

        NewThread ob1 = new NewThread("One", groupA);
        NewThread ob2 = new NewThread("Two", groupA);
        NewThread ob3 = new NewThread("Three", groupB);
        NewThread ob4 = new NewThread("Four", groupB);

        System.out.println("\nHere is output from list():");
        groupA.list();
        groupB.list();
        System.out.println();
Sample output from this program is shown here (the precise output you see may differ):

```
New thread: Thread[One,5,Group A]
New thread: Thread[Two,5,Group A]
New thread: Thread[Three,5,Group B]
New thread: Thread[Four,5,Group B]
Here is output from list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
    Thread[One,5,Group A]
    Thread[Two,5,Group A]
java.lang.ThreadGroup[name=Group B,maxpri=10]
    Thread[Three,5,Group B]
    Thread[Four,5,Group B]
```
Inside the program, notice that thread group A is suspended for four seconds. As the output confirms, this causes threads One and Two to pause, but threads Three and Four continue running. After the four seconds, threads One and Two are resumed. Notice how thread group A is suspended and resumed. First, the threads in group A are obtained by calling enumerate( ) on group A. Then, each thread is suspended by iterating through the resulting array. To resume the threads in A, the list is again traversed and each thread is resumed. One last point: This example uses the recommended approach to suspending and resuming threads. It does not rely upon
the deprecated methods suspend() and resume().

**ThreadLocal and InheritableThreadLocal**

Java defines two additional thread-related classes in java.lang:

- **ThreadLocal** Used to create thread local variables. Each thread will have its own copy of a thread local variable.
- **InheritableThreadLocal** Creates thread local variables that may be inherited.

**Package**

Package encapsulates version data associated with a package. Package version information is becoming more important because of the proliferation of packages and because a Java program may need to know what version of a package is available. The methods defined by Package are shown in Table 16-19. The following program demonstrates Package, displaying the packages about which the program currently is aware:

```java
// Demonstrate Package
class PkgTest {
    public static void main(String args[]) {
        Package pkgs[];

        pkgs = Package.getPackages();

        for(int i=0; i < pkgs.length; i++)
            System.out.println(
                pkgs[i].getName() + " " +
                pkgs[i].getImplementationTitle() + " " +
                pkgs[i].getImplementationVendor() + " " +
                pkgs[i].getImplementationVersion()
            );
    }
}
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;A extends Annotation&gt; A getAnnotation(Class&lt;A&gt; annoType)</code></td>
<td>Returns an <code>Annotation</code> object that contains the annotation associated with <code>annoType</code> for the invoking object.</td>
</tr>
<tr>
<td><code>Annotation[ ] getAnnotations()</code></td>
<td>Returns all annotations associated with the invoking object in an array of <code>Annotation</code> objects. Returns a reference to this array.</td>
</tr>
<tr>
<td><code>Annotation[ ] getDeclaredAnnotations()</code></td>
<td>Returns an <code>Annotation</code> object for all the annotations that are declared by the invoking object. (Inherited annotations are ignored.)</td>
</tr>
<tr>
<td><code>String getImplementationTitle()</code></td>
<td>Returns the title of the invoking package.</td>
</tr>
<tr>
<td><code>String getImplementationVendor()</code></td>
<td>Returns the name of the implementor of the invoking package.</td>
</tr>
<tr>
<td><code>String getImplementationVersion()</code></td>
<td>Returns the version number of the invoking package.</td>
</tr>
<tr>
<td><code>String getName()</code></td>
<td>Returns the name of the invoking package.</td>
</tr>
<tr>
<td><code>static Package getPackage(String pkgName)</code></td>
<td>Returns a <code>Package</code> object with the name specified by <code>pkgName</code>.</td>
</tr>
<tr>
<td><code>static Package[ ] getPackages()</code></td>
<td>Returns all packages about which the invoking program is currently aware.</td>
</tr>
<tr>
<td><code>String getSpecificationTitle()</code></td>
<td>Returns the title of the invoking package’s specification.</td>
</tr>
<tr>
<td><code>String getSpecificationVendor()</code></td>
<td>Returns the name of the owner of the specification for the invoking package.</td>
</tr>
<tr>
<td><code>String getSpecificationVersion()</code></td>
<td>Returns the invoking package’s specification version number.</td>
</tr>
<tr>
<td><code>int hashCode()</code></td>
<td>Returns the hash code for the invoking package.</td>
</tr>
<tr>
<td><code>boolean isAnnotationPresent(Class&lt;? extends Annotation&gt; anno)</code></td>
<td>Returns <code>true</code> if the annotation described by <code>anno</code> is associated with the invoking object. Returns <code>false</code>, otherwise.</td>
</tr>
<tr>
<td><code>boolean isCompatibleWith(String verNum)</code></td>
<td>Returns <code>true</code> if <code>verNum</code> is less than or equal to the invoking package’s version number.</td>
</tr>
<tr>
<td><code>boolean isSealed()</code></td>
<td>Returns <code>true</code> if the invoking package is sealed. Returns <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>boolean isSealed(URL url)</code></td>
<td>Returns <code>true</code> if the invoking package is sealed relative to <code>url</code>. Returns <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>String toString()</code></td>
<td>Returns the string equivalent of the invoking package.</td>
</tr>
</tbody>
</table>

**Table 16-19 The Methods Defined by Package**

**RuntimePermission**

RuntimePermission relates to Java’s security mechanism and is not examined further here.
Throwable

The Throwable class supports Java’s exception-handling system and is the class from which all exception classes are derived. It is discussed in Chapter 10.

SecurityManager

SecurityManager is a class that your classes can subclass to create a security manager. Generally, you don’t need to implement your own security manager. If you do, you need to consult the documentation that comes with your Java development system.

StackTraceElement

The StackTraceElement class describes a single stack frame, which is an individual element of a stack trace when an exception occurs. Each stack frame represents an execution point, which includes such things as the name of the class, the name of the method, the name of the file, and the source-code line number. An array of StackTraceElements is returned by the getStackTrace() method of the Throwable class.

StackTraceElement has one constructor:

```
StackTraceElement(String className, String methName, string fileName, int line)
```

Here, the name of the class is specified by className, the name of the method is specified in methName, the name of the file is specified by fileName, and the line number is passed in line. If there is no valid line number, use a negative value for line. Furthermore, a value of −2 for
The methods supported by StackTraceElement are shown in Table 16-20. These methods give you programmatic access to a stack trace.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object ob)</td>
<td>Returns true if the invoking StackTraceElement is the same as the one passed in ob. Otherwise, it returns false.</td>
</tr>
<tr>
<td>String getClassName()</td>
<td>Returns the name of the class in which the execution point described by the invoking StackTraceElement occurred.</td>
</tr>
<tr>
<td>String getFileName()</td>
<td>Returns the name of the file in which the source code of the execution point described by the invoking StackTraceElement is stored.</td>
</tr>
<tr>
<td>int getLineNumber()</td>
<td>Returns the source-code line number at which the execution point described by the invoking StackTraceElement occurred. In some situations, the line number will not be available, in which case a negative value is returned.</td>
</tr>
<tr>
<td>String getMethodName()</td>
<td>Returns the name of the method in which the execution point described by the invoking StackTraceElement occurred.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns the hash code for the invoking StackTraceElement.</td>
</tr>
<tr>
<td>boolean isNativeMethod()</td>
<td>Returns true if the execution point described by the invoking StackTraceElement occurred in a native method. Otherwise, it returns false.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns the String equivalent of the invoking sequence.</td>
</tr>
</tbody>
</table>

Table 16-20 The Methods Defined by StackTraceElement

Enum

As described in Chapter 12, an enumeration is a list of named constants. (Recall that an enumeration is created by using the keyword enum.) All enumerations automatically inherit Enum. Enum is a generic class that is declared as shown here:

```java
class Enum<E extends Enum<E>>
```

Here, E stands for the enumeration type. Enum has no public constructors.
Enum defines several methods that are available for use by all enumerations, which are shown in Table 16-21.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected final Object clone( ) throws CloneNotSupportedException</td>
<td>Invoking this method causes a CloneNotSupportedException to be thrown. This prevents enumerations from being cloned.</td>
</tr>
<tr>
<td>final int compareTo(E e)</td>
<td>Compares the ordinal value of two constants of the same enumeration. Returns a negative value if the invoking constant has an ordinal value less than e’s, zero if the two ordinal values are the same, and a positive value if the invoking constant has an ordinal value greater than e’s.</td>
</tr>
<tr>
<td>final boolean equals(Object obj)</td>
<td>Returns true if obj and the invoking object refer to the same constant.</td>
</tr>
<tr>
<td>final Class&lt;E&gt; getDeclaringClass( )</td>
<td>Returns the type of enumeration of which the invoking constant is a member.</td>
</tr>
<tr>
<td>final int hashCode( )</td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td>final String name( )</td>
<td>Returns the unaltered name of the invoking constant.</td>
</tr>
<tr>
<td>final int ordinal( )</td>
<td>Returns a value that indicates an enumeration constant’s position in the list of constants.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns the name of the invoking constant. This name may differ from the one used in the enumeration’s declaration.</td>
</tr>
<tr>
<td>static &lt;T extends Enum&lt;T&gt;&gt; T valueOf(Class&lt;T&gt; e-type, String name)</td>
<td>Returns the constant associated with name in the enumeration type specified by e-type.</td>
</tr>
</tbody>
</table>

Table 16-21 The Methods Defined by Enum

ClassValue

Added by JDK 7, ClassValue can be used to associate a value with a type. It is a generic type defined like this:

Class ClassValue<T>

It is designed for highly specialized uses, not for normal programming.

The CharSequence Interface

The CharSequence interface defines methods that grant
read-only access to a sequence of characters. These methods are shown in Table 16-22. This interface is implemented by String, StringBuffer, and StringBuilder, among others.

The Comparable Interface
Objects of classes that implement Comparable can be ordered. In other words, classes that implement Comparable contain objects that can be compared in some meaningful manner. Comparable is generic and is declared like this:

    interface Comparable<T>

Here, T represents the type of objects being compared.

    The Comparable interface declares one method that is used to determine what Java calls the natural ordering of instances of a class. The signature of the method is shown here:

    int compareTo(T obj)

This method compares the invoking object with obj. It returns 0 if the values are equal. A negative value is returned if the invoking object has a lower value. Otherwise, a positive value is returned.

    This interface is implemented by several of the classes already reviewed in this book. Specifically, the Byte, Character, Double, Float, Long, Short, String, and Integer classes define a compareTo() method. So does Enum.
The Appendable Interface

Objects of a class that implements Appendable can have a character or character sequences appended to it. Appendable defines these three methods:

- `Appendable append(char ch) throws IOException`
- `Appendable append(CharSequence chars) throws IOException`
- `Appendable append(CharSequence chars, int begin, int end) throws IOException`

In the first form, the character `ch` is appended to the invoking object. In the second form, the character sequence `chars` is appended to the invoking object. The third form allows you to indicate a portion (the characters running from `begin` through `end-1`) of the sequence specified by `chars`. In all cases, a reference to the invoking object is returned.

The Iterable Interface

Iterable must be implemented by any class whose objects will be used by the for-each version of the for loop. In other words, in order for an object to be used within a for-each style for loop, its class must implement Iterable.
Iterable is a generic interface that has this declaration:

```java
interface Iterable<T>
```

Here, T is the type of the object being iterated. It defines one method, `iterator()`, which is shown here:

```java
Iterator<T> iterator()
```

It returns an iterator to the elements contained in the invoking object.

**NOTE**

Iterators are described in detail in [Chapter 17](#).

The Readable Interface

The `Readable` interface indicates that an object can be used as a source for characters. It defines one method called `read()`, which is shown here:

```java
int read(CharBuffer buf) throws IOException
```

This method reads characters into `buf`. It returns the number of characters read, or -1 if an EOF is encountered.

The AutoCloseable Interface

`AutoCloseable` was added by JDK 7, and it provides support for the new `try-with-resources` statement, which implements what is sometimes referred to as automatic resource management (ARM). The `try-with-resources` statement automates the process of releasing a resource (such as a stream) when it is no longer needed. (See [Chapter 13](#) for details.) Only objects of classes that
implement AutoCloseable can be used with try-with-resources. The AutoCloseable interface defines only the close( ) method, which is shown here:

```java
void close( ) throws Exception
```

This method closes the invoking object, releasing any resources that it may hold. It is automatically called at the end of a try-with-resources statement, thus eliminating the need to explicitly invoke close( ).

AutoCloseable is implemented by several classes, including all of the I/O classes that open a stream that can be closed.

The Thread.UncaughtExceptionHandler Interface

The static Thread.UncaughtExceptionHandler interface is implemented by classes that want to handle uncaught exceptions. It is implemented by ThreadGroup. It declares only one method, which is shown here:

```java
void uncaughtException(Thread thrd, Throwable exc)
```

Here, thrd is a reference to the thread that generated the exception and exc is a reference to the exception.

The java.lang Subpackages

Java defines several subpackages of java.lang:

- java.lang.annotation
- java.lang.instrument
- java.lang.invoke
- java.lang.management
- java.lang.ref
- java.lang.reflect

Each is briefly described here.

java.lang.annotation

Java’s annotation facility is supported by java.lang.annotation. It defines the Annotation interface, and the ElementType and RetentionPolicy enumerations. Annotations are described in Chapter 12.

java.lang.instrument

java.lang.instrument defines features that can be used to add instrumentation to various aspects of program execution. It defines the Instrumentation and ClassFileTransformer interfaces, and the ClassDefinition class.

java.lang.invoke

Added by JDK 7, java.lang.invoke supports dynamic languages. It includes classes such as CallSite, MethodHandle, and MethodType.

java.lang.management

The java.lang.management package provides management support for the JVM and the execution environment. Using the features in java.lang.management, you can observe and manage various aspects of program execution.

java.lang.ref

You learned earlier that the garbage collection facilities...
in Java automatically determine when no references exist to an object. The object is then assumed to be no longer needed and its memory is reclaimed. The classes in the java.lang.ref package provide more flexible control over the garbage collection process.

java.lang.reflect

Reflection is the ability of a program to analyze code at run time. The java.lang.reflect package provides the ability to obtain information about the fields, constructors, methods, and modifiers of a class. Among other reasons, you need this information to build software tools that enable you to work with Java Beans components. The tools use reflection to determine dynamically the characteristics of a component. Reflection was introduced in Chapter 12 and is also examined in Chapter 28.

java.lang.reflect defines several classes, including Method, Field, and Constructor. It also defines several interfaces, including AnnotatedElement, Member, and Type. In addition, the java.lang.reflect package includes the Array class that enables you to create and access arrays dynamically.
This chapter begins our examination of java.util. This important package contains a large assortment of classes and interfaces that support a broad range of functionality. For example, java.util has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data. The java.util package also contains one of Java’s most powerful subsystems: the Collections Framework. The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. It merits close attention by all programmers.

Because java.util contains a wide array of functionality, it is quite large. Here is a list of its top-level classes:
The interfaces defined by \texttt{java.util} are shown next:

<table>
<thead>
<tr>
<th>AbstractCollection</th>
<th>EventObject</th>
<th>PropertyResourceBundle</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractList</td>
<td>FormattableFlags</td>
<td>Random</td>
</tr>
<tr>
<td>AbstractMap</td>
<td>Formatter</td>
<td>ResourceBundle</td>
</tr>
<tr>
<td>AbstractQueue</td>
<td>GregorianCalendar</td>
<td>Scanner</td>
</tr>
<tr>
<td>AbstractSequentialList</td>
<td>HashMap</td>
<td>ServiceLoader</td>
</tr>
<tr>
<td>AbstractSet</td>
<td>HashSet</td>
<td>SimpleTimeZone</td>
</tr>
<tr>
<td>ArrayDeque</td>
<td>Hashtable</td>
<td>Stack</td>
</tr>
<tr>
<td>ArrayList</td>
<td>IdentityHashMap</td>
<td>StringTokenizer</td>
</tr>
<tr>
<td>Arrays</td>
<td>LinkedHashMap</td>
<td>Timer</td>
</tr>
<tr>
<td>BitSet</td>
<td>LinkedHashSet</td>
<td>TimerTask</td>
</tr>
<tr>
<td>Calendar</td>
<td>LinkedList</td>
<td>TimeZone</td>
</tr>
<tr>
<td>Collections</td>
<td>ListResourceBundle</td>
<td>TreeMap</td>
</tr>
<tr>
<td>Currency</td>
<td>Locale</td>
<td>TreeSet</td>
</tr>
<tr>
<td>Date</td>
<td>Objects (Added by JDK 7.)</td>
<td>UUID</td>
</tr>
<tr>
<td>Dictionary</td>
<td>Observable</td>
<td>Vector</td>
</tr>
<tr>
<td>EnumMap</td>
<td>PriorityQueue</td>
<td>WeakHashMap</td>
</tr>
<tr>
<td>EnumSet</td>
<td>Properties</td>
<td></td>
</tr>
<tr>
<td>EventListenerProxy</td>
<td>PropertyPermission</td>
<td></td>
</tr>
</tbody>
</table>

Because of its size, the description of \texttt{java.util} is broken into two chapters. This chapter examines those members of \texttt{java.util} that are part of the Collections Framework. \textbf{Chapter 18} discusses its other classes and interfaces.

\textbf{Collections Overview}

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. The way that you used Vector was different from the way that you used Properties, for example. Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections are an answer to these (and other) problems.

The Collections Framework was designed to meet several goals. First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient. You seldom, if ever, need to code one of these “data engines” manually. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces. Several standard implementations (such as LinkedList, HashSet, and TreeSet) of these interfaces are provided that you may use as-is. You may also implement your own collection, if you choose. Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class
easier. Finally, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the Collections class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.

Another item closely associated with the Collections Framework is the Iterator interface. An iterator offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of enumerating the contents of a collection. Because each collection provides an iterator, the elements of any collection class can be accessed through the methods defined by Iterator. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are part of the Collections Framework, they are not “collections” in the strict use of the term. You can, however, obtain a collection-view of a map. Such a view contains the elements from the map stored in a collection. Thus, you can process the contents of a map as a collection, if you choose.

The collection mechanism was retrofitted to some of
the original classes defined by java.util so that they too could be integrated into the new system. It is important to understand that although the addition of collections altered the architecture of many of the original utility classes, it did not cause the deprecation of any. Collections simply provide a better way of doing several things.

NOTE
If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

JDK 5 Changed the Collections Framework
When JDK 5 was released, some fundamental changes were made to the Collections Framework that significantly increased its power and streamlined its use. These changes include the addition of generics, autoboxing/unboxing, and the for-each style for loop. Although JDK 7 is two major Java releases after JDK 5, the effects of the JDK 5 features were so profound that they still warrant special attention. The main reason is that much pre-JDK 5 code is still in use. Understanding the effects and reasons for the changes is important if you will be maintaining or updating older code.
Generics Fundamentally Changed the Collections Framework

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework was reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type parameters. Simply put, the addition of generics affected every part of the Collections Framework.

Generics added the one feature that collections had been missing: type safety. Prior to generics, all collections stored Object references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store incompatible types in a collection. Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

Although the addition of generics changed the declarations of most of its classes and interfaces, and several of their methods, overall, the Collections Framework still works the same as it did prior to generics. However, if you are familiar with the pre-generics version of the Collections Framework, you might find the new syntax a bit intimidating. Don’t worry; over time, the generic syntax will become second nature.

One other point: to gain the advantages that generics bring collections, older code will need to be rewritten.
This is also important because pre-generics code will generate warning messages when compiled by a modern Java compiler. To eliminate these warnings, you will need to add type information to all your collections code.

Autoboxing Facilitates the Use of Primitive Types

Autoboxing/unboxing facilitates the storing of primitive types in collections. As you will see, a collection can store only references, not primitive values. In the past, if you wanted to store a primitive value, such as an int, in a collection, you had to manually box it into its type wrapper. When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type. Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

The For-Each Style for Loop

All collection classes in the Collections Framework were retrofitted to implement the Iterable interface, which means that a collection can be cycled through by use of the for-each style for loop. In the past, cycling through a collection required the use of an iterator (described later in this chapter), with the programmer manually constructing the loop. Although iterators are still needed for some uses, in many cases, iterator-based loops can be replaced by for loops.
The Collection Interfaces

The Collections Framework defines several interfaces. This section provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes. Put differently, the concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized in the following table:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection</td>
<td>Enables you to work with groups of objects; it is at the top of the collections hierarchy.</td>
</tr>
<tr>
<td>Deque</td>
<td>Extends Queue to handle a double-ended queue.</td>
</tr>
<tr>
<td>List</td>
<td>Extends Collection to handle sequences (lists of objects).</td>
</tr>
<tr>
<td>NavigableSet</td>
<td>Extends SortedSet to handle retrieval of elements based on closest-match searches.</td>
</tr>
<tr>
<td>Queue</td>
<td>Extends Collection to handle special types of lists in which elements are removed only from the head.</td>
</tr>
<tr>
<td>Set</td>
<td>Extends Collection to handle sets, which must contain unique elements.</td>
</tr>
<tr>
<td>SortedSet</td>
<td>Extends Set to handle sorted sets.</td>
</tr>
</tbody>
</table>

In addition to the collection interfaces, collections also use the Comparator, RandomAccess, Iterator, and ListIterator interfaces, which are described in depth later in this chapter. Briefly, Comparator defines how two objects are compared; Iterator and ListIterator enumerate the objects within a collection. By implementing RandomAccess, a list indicates that it supports efficient, random access to its elements.

To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify the contents of a collection. Collections that support these methods
are called modifiable. Collections that do not allow their contents to be changed are called unmodifiable. If an attempt is made to use one of these methods on an unmodifiable collection, an UnsupportedOperationException is thrown. All the built-in collections are modifiable.

The following sections examine the collection interfaces.

The Collection Interface

The Collection interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection. Collection is a generic interface that has this declaration:

```
interface Collection<E>
```

Here, E specifies the type of objects that the collection will hold. Collection extends the Iterable interface. This means that all collections can be cycled through by use of the for-each style for loop. (Recall that only classes that implement Iterable can be cycled through by the for.)

Collection declares the core methods that all collections will have. These methods are summarized in Table 17-1. Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an UnsupportedOperationException. As explained, this occurs if a collection cannot be
modiﬁed. A ClassCastException is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the collection. An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a ﬁxed-length collection that is full.

Objects are added to a collection by calling add( ). Notice that add( ) takes an argument of type E, which means that objects added to a collection must be compatible with the type of data expected by the collection. You can add the entire contents of one collection to another by calling addAll( ).

You can remove an object by using remove( ). To remove a group of objects, call removeAll( ). You can remove all elements except those of a speciﬁed group by calling retainAll( ). To empty a collection, call clear( ).

You can determine whether a collection contains a speciﬁc object by calling contains( ). To determine whether one collection contains all the members of another, call containsAll( ). You can determine when a collection is empty by calling isEmpty( ). The number of elements currently held in a collection can be determined by calling size( ).
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean add(E obj)</td>
<td>Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection and the collection does not allow duplicates.</td>
</tr>
<tr>
<td>boolean addAll(Collection&lt;? extends E&gt; c)</td>
<td>Adds all the elements of c to the invoking collection. Returns true if the collection changed (i.e., the elements were added). Otherwise, returns false.</td>
</tr>
<tr>
<td>void clear()</td>
<td>Removes all elements from the invoking collection.</td>
</tr>
<tr>
<td>boolean contains(Object obj)</td>
<td>Returns true if obj is an element of the invoking collection. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean containsAll(Collection&lt;?&gt; c)</td>
<td>Returns true if the invoking collection contains all elements of c. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean equals(Object obj)</td>
<td>Returns true if the invoking collection and obj are equal. Otherwise, returns false.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns the hash code for the invoking collection.</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Returns true if the invoking collection is empty. Otherwise, returns false.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt; iterator()</td>
<td>Returns an iterator for the invoking collection.</td>
</tr>
<tr>
<td>boolean remove(Object obj)</td>
<td>Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean removeAll(Collection&lt;?&gt; c)</td>
<td>Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean retainAll(Collection&lt;?&gt; c)</td>
<td>Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the number of elements held in the invoking collection.</td>
</tr>
<tr>
<td>Object[] toArray()</td>
<td>Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.</td>
</tr>
<tr>
<td>&lt;T&gt; T[] toArray(T[] array)</td>
<td>Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of array equals the number of elements, these are returned in array. If the size of array is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of array is greater than the number of elements, the array element following the last collection element is set to null. An ArrayStoreException is thrown if any collection element has a type that is not a subtype of array.</td>
</tr>
</tbody>
</table>

Table 17-1 The Methods Defined by Collection
The `toArray()` methods return an array that contains the elements stored in the invoking collection. The first returns an array of `Object`. The second returns an array of elements that have the same type as the array specified as a parameter. Normally, the second form is more convenient because it returns the desired array type. These methods are more important than it might at first seem. Often, processing the contents of a collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you can have the best of both worlds.

Two collections can be compared for equality by calling `equals()`. The precise meaning of “equality” may differ from collection to collection. For example, you can implement `equals()` so that it compares the values of elements stored in the collection. Alternatively, `equals()` can compare references to those elements.

One more very important method is `iterator()`, which returns an iterator to a collection. Iterators are frequently used when working with collections.

The List Interface

The `List` interface extends `Collection` and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements. `List` is a generic interface that has this declaration:

```java
interface List<E>
```
Here, E specifies the type of objects that the list will hold.

In addition to the methods defined by Collection, List defines some of its own, which are summarized in Table 17-2. Note again that several of these methods will throw an UnsupportedOperationException if the list cannot be modified, and a ClassCastException is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a list. Also, several methods will throw an IndexOutOfBoundsException if an invalid index is used. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the list. An IllegalArgumentException is thrown if an invalid argument is used.

To the versions of add( ) and addAll( ) defined by Collection, List adds the methods add(int, E) and addAll(int, Collection). These methods insert elements at the specified index. Also, the semantics of add(E) and addAll(Collection) defined by Collection are changed by List so that they add elements to the end of the list.

To obtain the object stored at a specific location, call get( ) with the index of the object. To assign a value to an element in the list, call set( ), specifying the index of the object to be changed. To find the index of an object, use indexOf( ) or lastIndexOf( ).

You can obtain a sublist of a list by calling subList( ), specifying the beginning and ending indexes of the sublist. As you can imagine, subList( ) makes list
The Set Interface

The Set interface defines a set. It extends Collection and declares the behavior of a collection that does not allow duplicate elements. Therefore, the add() method returns false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own. Set is a generic interface that has this declaration:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(int index, E obj)</td>
<td>Inserts obj into the invoking list at the index passed in index. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.</td>
</tr>
<tr>
<td>boolean addAll(int index, Collection&lt;? extends E&gt; c)</td>
<td>Inserts all elements of c into the invoking list at the index passed in index. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.</td>
</tr>
<tr>
<td>E get(int index)</td>
<td>Returns the object stored at the specified index within the invoking collection.</td>
</tr>
<tr>
<td>int indexOf(Object obj)</td>
<td>Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, −1 is returned.</td>
</tr>
<tr>
<td>int lastIndexOf(Object obj)</td>
<td>Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, −1 is returned.</td>
</tr>
<tr>
<td>ListIterator&lt;E&gt; listIterator()</td>
<td>Returns an iterator to the start of the invoking list.</td>
</tr>
<tr>
<td>ListIterator&lt;E&gt; listIterator(int index)</td>
<td>Returns an iterator to the invoking list that begins at the specified index.</td>
</tr>
<tr>
<td>E remove(int index)</td>
<td>Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.</td>
</tr>
<tr>
<td>E set(int index, E obj)</td>
<td>Assigns obj to the location specified by index within the invoking list. Returns the old value.</td>
</tr>
<tr>
<td>List&lt;E&gt; subList(int start, int end)</td>
<td>Returns a list that includes elements from start to end−1 in the invoking list. Elements in the returned list are also referenced by the invoking object.</td>
</tr>
</tbody>
</table>

Table 17-2 The Methods Defined by List
interface Set\(<E>\>

Here, \(E\) specifies the type of objects that the set will hold.

The SortedSet Interface

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order. SortedSet is a generic interface that has this declaration:

```java
interface SortedSet\(<E>\>
```

Here, \(E\) specifies the type of objects that the set will hold.

In addition to those methods provided by Set, the SortedSet interface declares the methods summarized in Table 17-3. Several methods throw a NoSuchElementException when no items are contained in the invoking set. A ClassCastException is thrown when an object is incompatible with the elements in a set. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set. An IllegalArgumentException is thrown if an invalid argument is used.

SortedSet defines several methods that make set processing more convenient. To obtain the first object in the set, call `first( )`. To get the last element, use `last( )`. You can obtain a subset of a sorted set by calling `subSet( )`, specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use `headSet( )`. If you want the subset that ends the
The NavigableSet Interface

The NavigableSet interface extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values. NavigableSet is a generic interface that has this declaration:

```java
interface NavigableSet<E>
```

Here, E specifies the type of objects that the set will hold. In addition to the methods that it inherits from SortedSet, NavigableSet adds those summarized in Table 17-4. A ClassCastException is thrown when an object is incompatible with the elements in the set. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set. An IllegalArgumentException is thrown if an invalid argument is used.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparator&lt;? super E&gt; comparator()</td>
<td>Returns the invoking sorted set’s comparator. If the natural ordering is used for this set, null is returned.</td>
</tr>
<tr>
<td>E first( )</td>
<td>Returns the first element in the invoking sorted set.</td>
</tr>
<tr>
<td>SortedSet&lt;E&gt; headSet(E end)</td>
<td>Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.</td>
</tr>
<tr>
<td>E last( )</td>
<td>Returns the last element in the invoking sorted set.</td>
</tr>
<tr>
<td>SortedSet&lt;E&gt; subSet(E start, E end)</td>
<td>Returns a SortedSet that includes those elements between start and end-1. Elements in the returned collection are also referenced by the invoking object.</td>
</tr>
<tr>
<td>SortedSet&lt;E&gt; tailSet(E start)</td>
<td>Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.</td>
</tr>
</tbody>
</table>
### Table 17-3 The Methods Defined by SortedSet

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E ceiling(E. obj)</td>
<td>Searches the set for the smallest element $e$ such that $e &gt;= obj$. If such an element is found, it is returned. Otherwise, null is returned.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt; descendingIterator( )</td>
<td>Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.</td>
</tr>
<tr>
<td>NavigableSet&lt;E&gt; descendingSet( )</td>
<td>Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set.</td>
</tr>
<tr>
<td>E floor(E. obj)</td>
<td>Searches the set for the largest element $e$ such that $e &lt;= obj$. If such an element is found, it is returned. Otherwise, null is returned.</td>
</tr>
<tr>
<td>NavigableSet&lt;E&gt; headSet(E.upperBound, boolean incl)</td>
<td>Returns a NavigableSet that includes all elements from the invoking set that are less than upperBound. If incl is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set.</td>
</tr>
<tr>
<td>E higher(E. obj)</td>
<td>Searches the set for the largest element $e$ such that $e &gt; obj$. If such an element is found, it is returned. Otherwise, null is returned.</td>
</tr>
<tr>
<td>E lower(E. obj)</td>
<td>Searches the set for the largest element $e$ such that $e &lt; obj$. If such an element is found, it is returned. Otherwise, null is returned.</td>
</tr>
<tr>
<td>E pollFirst( )</td>
<td>Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty.</td>
</tr>
<tr>
<td>E pollLast( )</td>
<td>Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. null is returned if the set is empty.</td>
</tr>
<tr>
<td>NavigableSet&lt;E&gt; subSet(E.lowerBound, boolean lowIncl, E.upperBound, boolean highIncl)</td>
<td>Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound and less than upperBound. If lowIncl is true, then an element equal to lowerBound is included. If highIncl is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set.</td>
</tr>
<tr>
<td>NavigableSet&lt;E&gt; tailSet(E.lowerBound, boolean incl)</td>
<td>Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound. If incl is true, then an element equal to lowerBound is included. The resulting set is backed by the invoking set.</td>
</tr>
</tbody>
</table>

### Table 17-4 The Methods Defined by NavigableSet

The Queue Interface

The Queue interface extends Collection and declares the behavior of a queue, which is often a first-in, first-out
list. However, there are types of queues in which the ordering is based upon other criteria. Queue is a generic interface that has this declaration:

```java
interface Queue<E>
```

Here, E specifies the type of objects that the queue will hold. The methods defined by Queue are shown in Table 17-5.

Several methods throw a ClassCastException when an object is incompatible with the elements in the queue. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the queue. An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length queue that is full. A NoSuchElementException is thrown if an attempt is made to remove an element from an empty queue.

Despite its simplicity, Queue offers several points of interest. First, elements can only be removed from the head of the queue. Second, there are two methods that obtain and remove elements: poll() and remove(). The difference between them is that poll() returns null if the queue is empty, but remove() throws an exception. Third, there are two methods, element() and peek(), that obtain but don’t remove the element at the head of the queue. They differ only in that element() throws an exception if the queue is empty, but peek() returns null. Finally, notice that offer() only attempts to add an
element to a queue. Because some queues have a fixed length and might be full, offer() can fail.

The Deque Interface

The Deque interface extends Queue and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks. Deque is a generic interface that has this declaration:

```java
interface Deque<E>
```

Here, E specifies the type of objects that the deque will hold. In addition to the methods that it inherits from Queue, Deque adds those methods summarized in Table 17-6. Several methods throw a ClassCastException when an object is incompatible with the elements in the deque. A NullPointerException is thrown if an attempt is made to store a null object and null elements are not allowed in the deque. An IllegalArgumentException is thrown if an invalid argument is used. An IllegalStateException is thrown if an attempt is made to add an element to a fixed-length deque that is full. A NoSuchElementException is thrown if an attempt is made to remove an element from an empty deque.
### Table 17-5 The Methods Defined by Queue

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E element( )</td>
<td>Returns the element at the head of the queue. The element is not removed. It throws <strong>NoSuchElementException</strong> if the queue is empty.</td>
</tr>
<tr>
<td>boolean offer(E obj)</td>
<td>Attempts to add <em>obj</em> to the queue. Returns <strong>true</strong> if <em>obj</em> was added and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td>E peek( )</td>
<td>Returns the element at the head of the queue. It returns <strong>null</strong> if the queue is empty. The element is not removed.</td>
</tr>
<tr>
<td>E poll( )</td>
<td>Returns the element at the head of the queue, removing the element in the process. It returns <strong>null</strong> if the queue is empty.</td>
</tr>
<tr>
<td>E remove( )</td>
<td>Removes the element at the head of the queue, returning the element in the process. It throws <strong>NoSuchElementException</strong> if the queue is empty.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addFirst(E obj)</td>
<td>Adds <em>obj</em> to the head of the deque. Throws an <strong>IllegalStateException</strong> if a capacity-restricted deque is out of space.</td>
</tr>
<tr>
<td>void addLast(E obj)</td>
<td>Adds <em>obj</em> to the tail of the deque. Throws an <strong>IllegalStateException</strong> if a capacity-restricted deque is out of space.</td>
</tr>
<tr>
<td>Iterator&lt;E&gt; descendingIterator()</td>
<td>Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.</td>
</tr>
<tr>
<td>E getFirst( )</td>
<td>Returns the first element in the deque. The object is not removed from the deque. It throws <strong>NoSuchElementException</strong> if the deque is empty.</td>
</tr>
<tr>
<td>E getLast( )</td>
<td>Returns the last element in the deque. The object is not removed from the deque. It throws <strong>NoSuchElementException</strong> if the deque is empty.</td>
</tr>
<tr>
<td>boolean offerFirst(E obj)</td>
<td>Attempts to add <em>obj</em> to the head of the deque. Returns <strong>true</strong> if <em>obj</em> was added and <strong>false</strong> otherwise. Therefore, this method returns <strong>false</strong> when an attempt is made to add <em>obj</em> to a full, capacity-restricted deque.</td>
</tr>
<tr>
<td>boolean offerLast(E obj)</td>
<td>Attempts to add <em>obj</em> to the tail of the deque. Returns <strong>true</strong> if <em>obj</em> was added and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td>E peekFirst( )</td>
<td>Returns the element at the head of the deque. It returns <strong>null</strong> if the deque is empty. The object is not removed.</td>
</tr>
<tr>
<td>E peekLast( )</td>
<td>Returns the element at the tail of the deque. It returns <strong>null</strong> if the deque is empty. The object is not removed.</td>
</tr>
</tbody>
</table>
Notice that `Deque` includes the methods `push()` and `pop()`. These methods enable a Deque to function as a stack. Also, notice the `descendingIterator()` method. It returns an iterator that returns elements in reverse order. In other words, it returns an iterator that moves from the end of the collection to the start. A Deque implementation can be capacity-restricted, which means that only a limited number of elements can be added to the deque. When this is the case, an attempt to add an element to the deque can fail. Deque allows you to handle such a failure in two ways. First, methods such as `addFirst()` and `addLast()` throw an `IllegalStateException` if a capacity-restricted deque is full. Second, methods such as `offerFirst()` and `offerLast()` return false if the

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E pollFirst()</code></td>
<td>Returns the element at the head of the deque, removing the element in the process. It returns <code>null</code> if the deque is empty.</td>
</tr>
<tr>
<td><code>E pollLast()</code></td>
<td>Returns the element at the tail of the deque, removing the element in the process. It returns <code>null</code> if the deque is empty.</td>
</tr>
<tr>
<td><code>E pop()</code></td>
<td>Returns the element at the head of the deque, removing it in the process. It throws <code>NoSuchElementException</code> if the deque is empty.</td>
</tr>
<tr>
<td><code>void push(E obj)</code></td>
<td>Adds <code>obj</code> to the head of the deque. Throws an <code>IllegalStateException</code> if a capacity-restricted deque is out of space.</td>
</tr>
<tr>
<td><code>E removeFirst()</code></td>
<td>Returns the element at the head of the deque, removing the element in the process. It throws <code>NoSuchElementException</code> if the deque is empty.</td>
</tr>
<tr>
<td><code>boolean removeFirstOccurrence(Object obj)</code></td>
<td>Removes the first occurrence of <code>obj</code> from the deque. Returns <code>true</code> if successful and <code>false</code> if the deque did not contain <code>obj</code>.</td>
</tr>
<tr>
<td><code>E removeLast()</code></td>
<td>Returns the element at the tail of the deque, removing the element in the process. It throws <code>NoSuchElementException</code> if the deque is empty.</td>
</tr>
<tr>
<td><code>boolean removeLastOccurrence(Object obj)</code></td>
<td>Removes the last occurrence of <code>obj</code> from the deque. Returns <code>true</code> if successful and <code>false</code> if the deque did not contain <code>obj</code>.</td>
</tr>
</tbody>
</table>

Table 17-6 The Methods Defined by Deque
element cannot be added.

The Collection Classes

Now that you are familiar with the collection interfaces, you are ready to examine the standard classes that implement them. Some of the classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. As a general rule, the collection classes are not synchronized, but as you will see later in this chapter, it is possible to obtain synchronized versions.

The standard collection classes are summarized in the following table:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractCollection</td>
<td>Implements most of the Collection interface.</td>
</tr>
<tr>
<td>AbstractList</td>
<td>Extends AbstractCollection and implements most of the List interface.</td>
</tr>
<tr>
<td>AbstractQueue</td>
<td>Extends AbstractCollection and implements parts of the Queue interface.</td>
</tr>
<tr>
<td>AbstractSequentialList</td>
<td>Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.</td>
</tr>
<tr>
<td>LinkedList</td>
<td>Implements a linked list by extending AbstractSequentialList.</td>
</tr>
<tr>
<td>ArrayList</td>
<td>Implements a dynamic array by extending AbstractList.</td>
</tr>
<tr>
<td>ArrayDeque</td>
<td>Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface.</td>
</tr>
<tr>
<td>AbstractSet</td>
<td>Extends AbstractCollection and implements most of the Set interface.</td>
</tr>
<tr>
<td>EnumSet</td>
<td>Extends AbstractSet for use with enum elements.</td>
</tr>
<tr>
<td>HashSet</td>
<td>Extends AbstractSet for use with a hash table.</td>
</tr>
<tr>
<td>LinkedHashSet</td>
<td>Extends HashSet to allow insertion-order iterations.</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>Extends AbstractQueue to support a priority-based queue.</td>
</tr>
<tr>
<td>TreeSet</td>
<td>Implements a set stored in a tree. Extends AbstractSet.</td>
</tr>
</tbody>
</table>

The following sections examine the concrete collection classes and illustrate their use.

NOTE
In addition to the collection classes, several legacy classes, such as Vector, Stack, and Hashtable, have been reengineered to support collections. These are examined later in this chapter.

The ArrayList Class

The ArrayList class extends AbstractList and implements the List interface. ArrayList is a generic class that has this declaration:

```java
class ArrayList<E>
```

Here, E specifies the type of objects that the list will hold.

ArrayList supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines ArrayList. In essence, an ArrayList is a variable-length array of object references. That is, an ArrayList can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

NOTE

Dynamic arrays are also supported by the legacy class Vector, which is described later in this chapter.
ArrayList has the constructors shown here:

ArrayList( )
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection c. The third constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

The following program shows a simple use of ArrayList. An array list is created for objects of type String, and then several strings are added to it. (Recall that a quoted string is translated into a String object.) The list is then displayed. Some of the elements are removed and the list is displayed again.
The output from this program is shown here:

Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]

Notice that al starts out empty and grows as elements
are added to it. When elements are removed, its size is reduced.

In the preceding example, the contents of a collection are displayed using the default conversion provided by 
`toString( )`, which was inherited from `AbstractCollection`. Although it is sufficient for short, sample programs, you seldom use this method to display the contents of a real-world collection. Usually, you provide your own output routines. But, for the next few examples, the default output created by `toString( )` is sufficient.

Although the capacity of an `ArrayList` object increases automatically as objects are stored in it, you can increase the capacity of an `ArrayList` object manually by calling `ensureCapacity( )`. You might want to do this if you know in advance that you will be storing many more items in the collection than it can currently hold. By increasing its capacity once, at the start, you can prevent several reallocations later. Because reallocations are costly in terms of time, preventing unnecessary ones improves performance. The signature for `ensureCapacity( )` is shown here:

```java
void ensureCapacity(int cap)
```

Here, `cap` specifies the new minimum capacity of the collection.

Conversely, if you want to reduce the size of the array that underlies an `ArrayList` object so that it is precisely as large as the number of items that it is currently holding, call `trimToSize( )`, shown here:
void trimToSize( )

Obtaining an Array from an ArrayList

When working with ArrayList, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling toArray( ), which is defined by Collection. Several reasons exist why you might want to convert a collection into an array, such as:

- To obtain faster processing times for certain operations
- To pass an array to a method that is not overloaded to accept a collection
- To integrate collection-based code with legacy code that does not understand collections

Whatever the reason, converting an ArrayList to an array is a trivial matter.

As explained earlier, there are two versions of toArray( ), which are shown again here for your convenience:

```
object[] toArray( )
<T> T[] toArray(T array[ ])
```

The first returns an array of Object. The second returns an array of elements that have the same type as T. Normally, the second form is more convenient because it returns the proper type of array. The following program demonstrates its use:

```java
// Convert an ArrayList into an array.
import java.util.*;
```
class ArrayListToArray {
    public static void main(String args[]) {
        // Create an array list.
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Add elements to the array list.
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);

        System.out.println("Contents of al: " + al);

        // Get the array.
        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia);

        int sum = 0;

        // Sum the array.
        for(int i : ia) sum += i;

        System.out.println("Sum is: " + sum);
    }
}

The output from the program is shown here:

Contents of al: [1, 2, 3, 4]
Sum is: 10

The program begins by creating a collection of integers. Next, `toArray()` is called and it obtains an array of Integers. Then, the contents of that array are summed by use of a for-each style for loop.

There is something else of interest in this program. As you know, collections can store only references, not values of primitive types. However, autoboxing makes it possible to pass values of type `int` to `add( )` without having to manually wrap them within an `Integer`, as the program shows. Autoboxing causes them to be
automatically wrapped. In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

The LinkedList Class

The LinkedList class extends AbstractSequentialList and implements the List, Deque, and Queue interfaces. It provides a linked-list data structure. LinkedList is a generic class that has this declaration:

```java
class LinkedList<E>
```

Here, E specifies the type of objects that the list will hold. LinkedList has the two constructors shown here:

- `LinkedList( )`
- `LinkedList(Collection<? extends E> c)`

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection c.

Because LinkedList implements the Deque interface, you have access to the methods defined by Deque. For example, to add elements to the start of a list, you can use `addFirst( )` or `offerFirst( )`. To add elements to the end of the list, use `addLast( )` or `offerLast( )`. To obtain the first element, you can use `getFirst( )` or `peekFirst( )`. To obtain the last element, use `getLast( )` or `peekLast( )`. To remove the first element, use `removeFirst( )` or `pollFirst( )`. To remove the last element, use `removeLast( )` or `pollLast( )`. 
The following program illustrates LinkedList:

```java
// Demonstrate LinkedList.
import java.util.*;

class LinkedListDemo {
    public static void main(String args[]) {
        // Create a linked list.
        LinkedList<String> ll = new LinkedList<String>();

        // Add elements to the linked list.
        ll.add("F");
        ll.add("B");
        ll.add("D");
        ll.add("E");
        ll.add("C");
        ll.addLast("Z");
        ll.addFirst("A");

        ll.add(1, "A2");

        System.out.println("Original contents of ll: " + ll);
    }
}
```
The output from this program is shown here:

Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]

Because LinkedList implements the List interface, calls to add(E) append items to the end of the list, as do calls to addLast(). To insert items at a specific location, use the add(int, E) form of add(), as illustrated by the call to add(1, “A2”) in the example.

Notice how the third element in ll is changed by employing calls to get() and set(). To obtain the current value of an element, pass get() the index at which the element is stored. To assign a new value to that index, pass set() the index and its new value.
The HashSet Class

HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage. HashSet is a generic class that has this declaration:

    class HashSet < E >

Here, E specifies the type of objects that the set will hold.

As most readers likely know, a hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can’t directly index the hash table. The advantage of hashing is that it allows the execution time of add( ), contains( ), remove( ), and size( ) to remain constant even for large sets.

The following constructors are defined:

    HashSet( )
    HashSet(Collection<? extends E> c)
    HashSet(int capacity)
    HashSet(int capacity, float fillRatio)

The first form constructs a default hash set. The second
form initializes the hash set by using the elements of c. The third form initializes the capacity of the hash set to capacity. (The default capacity is 16.) The fourth form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used.

HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

It is important to note that HashSet does not guarantee the order of its elements, because the process of hashing doesn’t usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as TreeSet, is a better choice.

Here is an example that demonstrates HashSet:

```java
// Demonstrate HashSet.
import java.util.*;

class HashSetDemo {
    public static void main(String args[]) {
        // Create a hash set.
        HashSet<String> hs = new HashSet<String>();

        // Add elements to the hash set.
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
        hs.add("F");
    }
}
```
The following is the output from this program:

[D, E, F, A, B, C]

As explained, the elements are not stored in sorted order, and the precise output may vary.

The LinkedHashSet Class

The LinkedHashSet class extends HashSet and adds no members of its own. It is a generic class that has this declaration:

```java
class LinkedHashSet<E>
```

Here, E specifies the type of objects that the set will hold. Its constructors parallel those in HashSet.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set. That is, when cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted. This is also the order in which they are contained in the string returned by `toString()` when called on a LinkedHashSet object. To see the effect of LinkedHashSet, try substituting LinkedHashSet for HashSet in the preceding program. The output will be

[B, A, D, E, C, F]

which is the order in which the elements were inserted.
The TreeSet Class

TreeSet extends AbstractSet and implements the NavigableSet interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

```java
class TreeSet<E>
```

Here, E specifies the type of objects that the set will hold.

TreeSet has the following constructors:

```java
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> comp)
TreeSet(SortedSet<E> ss)
```

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of c. The third form constructs an empty tree set that will be sorted according to the comparator specified by comp. (Comparators are described later in this chapter.) The fourth form builds a tree set that contains the elements of ss.

Here is an example that demonstrates a TreeSet:
// Demonstrate TreeSet.
import java.util.*;

class TreeSetDemo {
    public static void main(String args[]) {
        // Create a tree set.
        TreeSet<String> ts = new TreeSet<String>();

        // Add elements to the tree set.
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        System.out.println(ts);
    }
}

The output from this program is shown here:

[A, B, C, D, E, F]

As explained, because TreeSet stores its elements in a tree, they are automatically arranged in sorted order, as the output confirms.

Because TreeSet implements the NavigableSet interface, you can use the methods defined by NavigableSet to retrieve elements of a TreeSet. For example, assuming the preceding program, the following statement uses subSet( ) to obtain a subset of ts that contains the elements between C (inclusive) and F (exclusive). It then displays the resulting set.

    System.out.println(ts.subSet("C", "F"));

The output from this statement is shown here:

[C, D, E]
You might want to experiment with the other methods defined by NavigableSet.

The PriorityQueue Class

PriorityQueue extends AbstractQueue and implements the Queue interface. It creates a queue that is prioritized based on the queue’s comparator. PriorityQueue is a generic class that has this declaration:

```java
class PriorityQueue<E>
```

Here, E specifies the type of objects stored in the queue. Priorities are dynamic, growing as necessary.

PriorityQueue defines the six constructors shown here:

```java
PriorityQueue()
PriorityQueue(int capacity)
PriorityQueue(int capacity, Comparator<? super E> comp)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
```

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in `c`. In all cases, the capacity grows automatically as elements are added.
If no comparator is specified when a PriorityQueue is constructed, then the default comparator for the type of data stored in the queue is used. The default comparator will order the queue in ascending order. Thus, the head of the queue will be the smallest value. However, by providing a custom comparator, you can specify a different ordering scheme. For example, when storing items that include a time stamp, you could prioritize the queue such that the oldest items are first in the queue.

You can obtain a reference to the comparator used by a PriorityQueue by calling its comparator( ) method, shown here:

```java
Comparator<? super E> comparator()
```

It returns the comparator. If natural ordering is used for the invoking queue, null is returned.

One word of caution: Although you can iterate through a PriorityQueue using an iterator, the order of that iteration is undefined. To properly use a PriorityQueue, you must call methods such as offer( ) and poll( ), which are defined by the Queue interface.

The ArrayDeque Class

The ArrayDeque class extends AbstractCollection and implements the Deque interface. It adds no methods of its own. ArrayDeque creates a dynamic array and has no capacity restrictions. (The Deque interface supports implementations that restrict capacity, but does not require such restrictions.) ArrayDeque is a generic class
that has this declaration:

    class ArrayDeque<E>

Here, E specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

    ArrayDeque( )
    ArrayDeque(int size)
    ArrayDeque(Collection<? extends E> c)

The first constructor builds an empty deque. Its starting capacity is 16. The second constructor builds a deque that has the specified initial capacity. The third constructor creates a deque that is initialized with the elements of the collection passed in c. In all cases, the capacity grows as needed to handle the elements added to the deque.

The following program demonstrates ArrayDeque by using it to create a stack:

    // Demonstrate ArrayDeque.
    import java.util.*;
    class ArrayDequeDemo {
        public static void main(String args[]) {
            // Create an array deque.
            ArrayDeque<String> adq = new ArrayDeque<String>();

            // Use an ArrayDeque like a stack.
            adq.push("A");
            adq.push("B");
            adq.push("D");
            adq.push("E");
            adq.push("F");
            System.out.print("Popping the stack: ");
        }
    }
while(adq.peek() != null)
    System.out.print(adq.pop() + " ");

System.out.println();
}
}

The output is shown here:

Popping the stack: F E D B A

The EnumSet Class

EnumSet extends AbstractSet and implements Set. It is specifically for use with keys of an enum type. It is a generic class that has this declaration:

    class EnumSet<E extends Enum<E>>

Here, E specifies the elements. Notice that E must extend Enum<E>, which enforces the requirement that the elements must be of the specified enum type.

EnumSet defines no constructors. Instead, it uses the factory methods shown in Table 17-7 to create objects. All methods can throw NullPointerException. The copyOf( ) and range( ) methods can also throw IllegalArgumentException. Notice that the of( ) method is overloaded a number of times. This is in the interest of efficiency. Passing a known number of arguments can be faster than using a vararg parameter when the number of arguments is small.
### Table 17-7 The Methods Defined by EnumSet

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| static `<E extends Enum<E>>
EnumSet<E> allOf(Class<E> l)`                               | Creates an EnumSet that contains the elements in the enumeration specified by `l`. |
| static `<E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> e)` | Creates an EnumSet that is comprised of those elements not stored in `e`.   |
| static `<E extends Enum<E>>
EnumSet<E> copyOf(EnumSet<E> c)`                                 | Creates an EnumSet from the elements stored in `c`.                         |
| static `<E extends Enum<E>>
EnumSet<E> copyOf(Collection<E> c)`                                | Creates an EnumSet from the elements stored in `c`.                         |
| static `<E extends Enum<E>>
EnumSet<E> noneOf(Class<E> l)`                                   | Creates an EnumSet that contains the elements that are not in the enumeration specified by `l`, which is an empty set by definition. |
| static `<E extends Enum<E>>
EnumSet<E> of(E v, E ... varargs)`                                      | Creates an EnumSet that contains `v` and zero or more additional enumeration values. |
| static `<E extends Enum<E>>
EnumSet<E> of(E v)`                                                          | Creates an EnumSet that contains `v`.                                      |
| static `<E extends Enum<E>>
EnumSet<E> of(E v1, E v2)`                                                       | Creates an EnumSet that contains `v1` and `v2`.                             |
| static `<E extends Enum<E>>
EnumSet<E> of(E v1, E v2, E v3)`                                                   | Creates an EnumSet that contains `v1` through `v3`.                         |
| static `<E extends Enum<E>>
EnumSet<E> of(E v1, E v2, E v3, E v4)`                                               | Creates an EnumSet that contains `v1` through `v4`.                         |
| static `<E extends Enum<E>>
EnumSet<E> of(E v1, E v2, E v3, E v4, E v5)`                                | Creates an EnumSet that contains `v1` through `v5`.                         |
| static `<E extends Enum<E>>
EnumSet<E> range(E start, E end)`                                                | Creates an EnumSet that contains the elements in the range specified by `start` and `end`. |

---

**Accessing a Collection via an Iterator**

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface. Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
Iterator and ListIterator are generic interfaces which are declared as shown here:

```java
interface Iterator<E>
interface ListIterator<E>
```

Here, E specifies the type of objects being iterated. The Iterator interface declares the methods shown in Table 17-8. The methods declared by ListIterator are shown in Table 17-9.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean hasNext( )</td>
<td>Returns true if there are more elements. Otherwise, returns false.</td>
</tr>
<tr>
<td>E next( )</td>
<td>Returns the next element. Throws NoSuchElementException if there is not a next element.</td>
</tr>
<tr>
<td>void remove( )</td>
<td>Removes the current element. Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).</td>
</tr>
</tbody>
</table>

Table 17-8 The Methods Defined by Iterator

In both cases, operations that modify the underlying collection are optional. For example, remove( ) will throw UnsupportedOperationException when used with a read-only collection. Various other exceptions are possible.

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:
Table 17-9 The Methods Defined by ListIterator

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(E obj)</td>
<td>Inserts obj into the list in front of the element that will be returned by the next call to next( ).</td>
</tr>
<tr>
<td>boolean hasNext( )</td>
<td>Returns true if there is a next element. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean hasPrevious( )</td>
<td>Returns true if there is a previous element. Otherwise, returns false.</td>
</tr>
<tr>
<td>E next( )</td>
<td>Returns the next element. A NoSuchElementException is thrown if there is not a next element.</td>
</tr>
<tr>
<td>int nextIndex( )</td>
<td>Returns the index of the next element. If there is not a next element, returns the size of the list.</td>
</tr>
<tr>
<td>E previous( )</td>
<td>Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.</td>
</tr>
<tr>
<td>int previousIndex( )</td>
<td>Returns the index of the previous element. If there is not a previous element, returns -1.</td>
</tr>
<tr>
<td>void remove( )</td>
<td>Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.</td>
</tr>
<tr>
<td>void set(E obj)</td>
<td>Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).</td>
</tr>
</tbody>
</table>

1. Obtain an iterator to the start of the collection by calling the collection’s iterator( ) method.

2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.

3. Within the loop, obtain each element by calling next( ).

For collections that implement List, you can also obtain an iterator by calling listIterator( ). As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, ListIterator is used just like Iterator.

The following example implements these steps, demonstrating both the Iterator and ListIterator
interfaces. It uses an ArrayList object, but the general principles apply to any type of collection. Of course, ListIterator is available only to those collections that implement the List interface.

```java
// Demonstrate iterators.
import java.util.*;

class IteratorDemo {
    public static void main(String args[]) {
        // Create an array list.
        ArrayList<String> al = new ArrayList<String>();

        // Add elements to the array list.
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");

        // Use iterator to display contents of al.
        System.out.print("Original contents of al: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```
Pay special attention to how the list is displayed in reverse. After the list is modified, litr points to the end of the list. (Remember, litr.hasNext() returns false when the end of the list has been reached.) To traverse the list in reverse, the program continues to use litr, but this time it checks to see whether it has a previous element. As long as it does, that element is obtained and displayed.

The For-Each Alternative to Iterators
If you won’t be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the for loop is often a more convenient alternative to cycling through a collection than is using an iterator. Recall that the for can cycle through any collection of objects that implement the `Iterable` interface. Because all of the collection classes implement this interface, they can all be operated upon by the for.

The following example uses a for loop to sum the contents of a collection:
The output from the program is shown here:

Original contents of vals: 1 2 3 4 5
Sum of values: 15

As you can see, the for loop is substantially shorter and simpler to use than the iterator-based approach. However, it can only be used to cycle through a collection in the forward direction, and you can’t modify the contents of the collection.
Storing User-Defined Classes in Collections

For the sake of simplicity, the foregoing examples have stored built-in objects, such as String or Integer, in a collection. Of course, collections are not limited to the storage of built-in objects. Quite the contrary. The power of collections is that they can store any type of object, including objects of classes that you create. For example, consider the following example that uses a LinkedList to store mailing addresses:

```java
// A simple mailing list example.
import java.util.*;

class Address {
    private String name;
    private String street;
    private String city;
    private String state;
    private String code;

    Address(String n, String s, String c, String st, String cd) {
        name = n;
        street = s;
        city = c;
        state = st;
        code = cd;
    }

    public String toString() {
        return name + "\n" + street + "\n" + city + " " + state + " " + code;
    }
}
```
The output from the program is shown here:

J.W. West  
11 Oak Ave  
Urbana IL 61801  

Ralph Baker  
1142 Maple Lane  
Mahomet IL 61853  

Tom Carlton  
867 Elm St  
Champaign IL 61820

Aside from storing a user-defined class in a collection, another important thing to notice about the preceding program is that it is quite short. When you consider that it sets up a linked list that can store, retrieve, and process mailing addresses in about 50 lines of code, the power of the Collections Framework begins to become apparent. As most readers know, if all of this functionality had to be coded manually, the program would be several times
longer. Collections offer off-the-shelf solutions to a wide variety of programming problems. You should use them whenever the situation presents itself.

The RandomAccess Interface

The RandomAccess interface contains no members. However, by implementing this interface, a collection signals that it supports efficient random access to its elements. Although a collection might support random access, it might not do so efficiently. By checking for the RandomAccess interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections. (You can use instanceof to determine if a class implements an interface.) RandomAccess is implemented by ArrayList and by the legacy Vector class, among others.

Working with Maps

A map is an object that stores associations between keys and values, or key/value pairs. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.

There is one key point about maps that is important to mention at the outset: they don’t implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop. Furthermore, you can’t obtain an iterator to a map.
However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the for loop or an iterator.

The Map Interfaces
Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>Maps unique keys to values.</td>
</tr>
<tr>
<td>Map.Entry</td>
<td>Describes an element (a key/value pair) in a map. This is an inner class of Map.</td>
</tr>
<tr>
<td>NavigableMap</td>
<td>Extends SortedMap to handle the retrieval of entries based on closest-match searches.</td>
</tr>
<tr>
<td>SortedMap</td>
<td>Extends Map so that the keys are maintained in ascending order.</td>
</tr>
</tbody>
</table>

Each interface is examined next, in turn.

The Map Interface
The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key. Map is generic and is declared as shown here:

    interface Map<K, V>

Here, K specifies the type of keys, and V specifies the type of values.

The methods declared by Map are summarized in Table 17-10. Several methods throw a ClassCastException when an object is incompatible with
the elements in a map. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map. An
UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map. An
IllegalArgumentException is thrown if an invalid argument is used.

Maps revolve around two basic operations: get() and put(). To put a value into a map, use put(), specifying the key and the value. To obtain a value, call get(), passing the key as an argument. The value is returned.

As mentioned earlier, although part of the Collections Framework, maps are not, themselves, collections because they do not implement the Collection interface. However, you can obtain a collection-view of a map. To do this, you can use the entrySet() method. It returns a Set that contains the elements in the map. To obtain a collection-view of the keys, use keySet(). To get a collection-view of the values, use values(). For all three collection-views, the collection is backed by the map. Changing one affects the other. Collection-views are the means by which maps are integrated into the larger Collections Framework.
Table 17-10 The Methods Defined by Map

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear()</td>
<td>Removes all key/value pairs from the invoking map.</td>
</tr>
<tr>
<td>boolean containsKey(Object k)</td>
<td>Returns true if the invoking map contains k as a key. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean containsValue(Object v)</td>
<td>Returns true if the map contains v as a value. Otherwise, returns false.</td>
</tr>
<tr>
<td>Set&lt;Map.Entry&lt;K, V&gt;&gt; entrySet()</td>
<td>Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. Thus, this method provides a set-view of the invoking map.</td>
</tr>
<tr>
<td>boolean equals(Object obj)</td>
<td>Returns true if obj is a Map and contains the same entries. Otherwise, returns false.</td>
</tr>
<tr>
<td>V get(Object k)</td>
<td>Returns the value associated with the key k. Returns null if the key is not found.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns the hash code for the invoking map.</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Returns true if the invoking map is empty. Otherwise, returns false.</td>
</tr>
<tr>
<td>Set&lt;K&gt; keySet()</td>
<td>Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.</td>
</tr>
<tr>
<td>V put(K k, V v)</td>
<td>Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.</td>
</tr>
<tr>
<td>void putAll(Map&lt;? extends K, ? extends V&gt; m)</td>
<td>Puts all the entries from m into this map.</td>
</tr>
<tr>
<td>V remove(Object k)</td>
<td>Removes the entry whose key equals k.</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the number of key/value pairs in the map.</td>
</tr>
<tr>
<td>Collection&lt;V&gt; values()</td>
<td>Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.</td>
</tr>
</tbody>
</table>

The SortedMap Interface

The SortedMap interface extends Map. It ensures that the entries are maintained in ascending order based on the keys. SortedMap is generic and is declared as shown here:

```
interface SortedMap<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values.
The methods declared by SortedMap are summarized in Table 17-11. Several methods throw a NoSuchElementException when no items are in the invoking map. A ClassCastException is thrown when an object is incompatible with the elements in a map. A NullPointerException is thrown if an attempt is made to use a null object when null is not allowed in the map. An IllegalArgumentException is thrown if an invalid argument is used.

Sorted maps allow very efficient manipulations of submaps (in other words, subsets of a map). To obtain a submap, use headMap( ), tailMap( ), or subMap( ). The submap returned by these methods is backed by the invoking map. Changing one changes the other. To get the first key in the set, call firstKey( ). To get the last key, use lastKey( ).

The NavigableMap Interface

The NavigableMap interface extends SortedMap and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. NavigableMap is a generic interface that has this declaration:

```java
interface NavigableMap <K,V>
```

Here, K specifies the type of the keys, and V specifies the type of the values associated with the keys. In addition to the methods that it inherits from SortedMap, NavigableMap adds those summarized in Table 17-12. Several methods throw a ClassCastException when an
A `NullPointerException` is thrown if an attempt is made to use a null object and null keys are not allowed in the set. A `IllegalArgumentException` is thrown if an invalid argument is used.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Comparator&lt;? super K&gt; comparator()</code></td>
<td>Returns the invoking sorted map’s comparator. If natural ordering is used for the invoking map, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>K firstKey()</code></td>
<td>Returns the first key in the invoking map.</td>
</tr>
<tr>
<td><code>SortedMap&lt;K, V&gt; headMap(K end)</code></td>
<td>Returns a sorted map for those map entries with keys that are less than <code>end</code>.</td>
</tr>
<tr>
<td><code>K lastKey()</code></td>
<td>Returns the last key in the invoking map.</td>
</tr>
<tr>
<td><code>SortedMap&lt;K, V&gt; subMap(K start, K end)</code></td>
<td>Returns a map containing those entries with keys that are greater than or equal to <code>start</code> and less than <code>end</code>.</td>
</tr>
<tr>
<td><code>SortedMap&lt;K, V&gt; tailMap(K start)</code></td>
<td>Returns a map containing those entries with keys that are greater than or equal to <code>start</code>.</td>
</tr>
</tbody>
</table>

Table 17-11 The Methods Defined by SortedMap

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Map.Entry&lt;K,V&gt; ceilingEntry(K obj)</code></td>
<td>Searches the map for the smallest key <code>k</code> such that <code>k &gt;= obj</code>. If such a key is found, its entry is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>K ceilingKey(K obj)</code></td>
<td>Searches the map for the smallest key <code>k</code> such that <code>k &gt;= obj</code>. If such a key is found, it is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>NavigableSet&lt;K&gt; descendingKeySet()</code></td>
<td>Returns a <code>NavigableSet</code> that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.</td>
</tr>
<tr>
<td><code>NavigableMap&lt;K,V&gt; descendingMap()</code></td>
<td>Returns a <code>NavigableMap</code> that is the reverse of the invoking map. The resulting map is backed by the invoking map.</td>
</tr>
<tr>
<td><code>Map.Entry&lt;K,V&gt; firstEntry()</code></td>
<td>Returns the first entry in the map. This is the entry with the least key.</td>
</tr>
<tr>
<td><code>Map.Entry&lt;K,V&gt; floorEntry(K obj)</code></td>
<td>Searches the map for the largest key <code>k</code> such that <code>k &lt;= obj</code>. If such a key is found, its entry is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>K floorKey(K obj)</code></td>
<td>Searches the map for the largest key <code>k</code> such that <code>k &lt;= obj</code>. If such a key is found, it is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>headMap(K upperBound, boolean incl)</code></td>
<td>Returns a <code>NavigableMap</code> that includes all entries from the invoking map that have keys that are less than <code>upperBound</code>. If <code>incl</code> is <code>true</code>, then an element equal to <code>upperBound</code> is included. The resulting map is backed by the invoking map.</td>
</tr>
<tr>
<td><code>Map.Entry&lt;K,V&gt; higherEntry(K obj)</code></td>
<td>Searches the set for the largest key <code>k</code> such that <code>k &gt; obj</code>. If such a key is found, its entry is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>K higherKey(K obj)</code></td>
<td>Searches the set for the largest key <code>k</code> such that <code>k &gt; obj</code>. If such a key is found, it is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>Map.Entry&lt;K,V&gt; lastEntry()</code></td>
<td>Returns the last entry in the map. This is the entry with the largest key.</td>
</tr>
<tr>
<td><code>Map.Entry&lt;K,V&gt; lowerEntry(K obj)</code></td>
<td>Searches the set for the largest key <code>k</code> such that <code>k &lt; obj</code>. If such a key is found, its entry is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>K lowerKey(K obj)</code></td>
<td>Searches the set for the largest key <code>k</code> such that <code>k &lt; obj</code>. If such a key is found, it is returned. Otherwise, <code>null</code> is returned.</td>
</tr>
<tr>
<td><code>NavigableSet&lt;K&gt; navigableKeySet()</code></td>
<td>Returns a <code>NavigableSet</code> that contains the keys in the invoking map. The resulting set is backed by the invoking map.</td>
</tr>
<tr>
<td><code>Map.Entry&lt;K,V&gt; pollFirstEntry()</code></td>
<td>Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. <code>null</code> is returned if the map is empty.</td>
</tr>
<tr>
<td><code>Map.Entry&lt;K,V&gt; pollLastEntry()</code></td>
<td>Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. <code>null</code> is returned if the map is empty.</td>
</tr>
<tr>
<td><code>NavigableMap&lt;K,V&gt; subMap(K lowerBound, boolean lowIncl, K upperBound boolean highIncl)</code></td>
<td>Returns a <code>NavigableMap</code> that includes all entries from the invoking map that have keys that are greater than <code>lowerBound</code> and less than <code>upperBound</code>. If <code>lowIncl</code> is <code>true</code>, then an element equal to <code>lowerBound</code> is included. If <code>highIncl</code> is <code>true</code>, then an element equal to <code>highIncl</code> is included. The resulting map is backed by the invoking map.</td>
</tr>
<tr>
<td><code>NavigableMap&lt;K,V&gt; tailMap(K lowerBound, boolean incl)</code></td>
<td>Returns a <code>NavigableMap</code> that includes all entries from the invoking map that have keys that are greater than <code>lowerBound</code>. If <code>incl</code> is <code>true</code>, then an element equal to <code>lowerBound</code> is included. The resulting map is backed by the invoking map.</td>
</tr>
</tbody>
</table>

Table 17-12 The Methods defined by NavigableMap

The Map.Entry Interface

The `Map.Entry` interface enables you to work with a map...
Recall that the `entrySet()` method declared by the `Map` interface returns a `Set` containing the map entries. Each of these set elements is a `Map.Entry` object. `Map.Entry` is generic and is declared like this:

```java
interface Map.Entry<K, V>
```

Here, `K` specifies the type of keys, and `V` specifies the type of values. Table 17-13 summarizes the methods declared by `Map.Entry`. Various exceptions are possible.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean equals(Object obj)</code></td>
<td>Returns <code>true</code> if <code>obj</code> is a <code>Map.Entry</code> whose key and value are equal to that of the invoking object.</td>
</tr>
<tr>
<td><code>K getKey()</code></td>
<td>Returns the key for this map entry.</td>
</tr>
<tr>
<td><code>V getValue()</code></td>
<td>Returns the value for this map entry.</td>
</tr>
<tr>
<td><code>int hashCode()</code></td>
<td>Returns the hash code for this map entry.</td>
</tr>
<tr>
<td><code>V setValue(V v)</code></td>
<td>Sets the value for this map entry to <code>v</code>. A <code>ClassCastException</code> is thrown if <code>v</code> is not the correct type for the map. An <code>IllegalArgumentException</code> is thrown if there is a problem with <code>v</code>. A <code>NullPointerException</code> is thrown if <code>v</code> is <code>null</code> and the map does not permit <code>null</code> keys. An <code>UnsupportedOperationException</code> is thrown if the map cannot be changed.</td>
</tr>
</tbody>
</table>

Table 17-13 The Methods Defined by `Map.Entry`

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:
Notice that AbstractMap is a superclass for all concrete map implementations.

WeakHashMap implements a map that uses “weak keys,” which allows an element in a map to be garbage-collected when its key is otherwise unused. This class is not discussed further here. The other map classes are described next.

The HashMap Class
The HashMap class extends AbstractMap and implements the Map interface. It uses a hash table to store the map. This allows the execution time of get() and put() to remain constant even for large sets. HashMap is a generic class that has this declaration:

```java
class HashMap<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values.

The following constructors are defined:

```java
HashMap()
HashMap(Map<? extends K, ? extends V> m)
HashMap(int capacity)
```
The first form constructs a default hash map. The second form initializes the hash map by using the elements of `m`. The third form initializes the capacity of the hash map to `capacity`. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for HashSet, described earlier. The default capacity is 16. The default fill ratio is 0.75.

HashMap implements Map and extends AbstractMap. It does not add any methods of its own.

You should note that a hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The following program illustrates HashMap. It maps names to account balances. Notice how a set-view is obtained and used.

```java
import java.util.*;

class HashMapDemo {
    public static void main(String args[]) {
        // Create a hash map.
        HashMap<String, Double> hm = new HashMap<String, Double>();

        // Put elements to the map
        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Tod Hall", new Double(99.22));
        hm.put("Ralph Smith", new Double(-19.08));

        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = hm.entrySet();
    }
}
```
// Display the set.
for (Map.Entry<String, Double> me : set) {
    System.out.print(me.getKey() + "\t: ");
    System.out.println(me.getValue());
}
System.out.println();

// Deposit 1000 into John Doe's account.
double balance = hm.get("John Doe");
hm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance: " +
    hm.get("John Doe");
}

Output from this program is shown here (the precise order may vary):

Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe's new balance: 4434.34

The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling entrySet(). The keys and values are displayed by calling the getKey() and getValue() methods that are defined by Map.Entry. Pay close attention to how the deposit is made into John Doe’s account. The put() method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe’s account is updated, the hash map will still contain just one “John
The TreeMap Class

The TreeMap class extends AbstractMap and implements the NavigableMap interface. It creates maps stored in a tree structure. A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order. TreeMap is a generic class that has this declaration:

```java
class TreeMap<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values.

The following TreeMap constructors are defined:

- `TreeMap()`: constructs an empty tree map that will be sorted by using the natural order of its keys.
- `TreeMap(Comparator<? super K> comp)`: constructs an empty tree-based map that will be sorted by using the Comparator comp. (Comparators are discussed later in this chapter.)
- `TreeMap(Map<? extends K, ? extends V> m)`: initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys.
- `TreeMap(SortedMap<K, ? extends V> sm)`: initializes a tree map with the entries from sm, which will be sorted in the same order as sm.
TreeMap has no map methods beyond those specified by the NavigableMap interface and the AbstractMap class.

The following program reworks the preceding example so that it uses TreeMap:

```java
import java.util.*;

class TreeMapDemo {
    public static void main(String args[]) {

        // Create a tree map.
        TreeMap<String, Double> tm = new TreeMap<String, Double>();

        // Put elements to the map.
        tm.put("John Doe", new Double(3434.34));
        tm.put("Tom Smith", new Double(123.22));
        tm.put("Jane Baker", new Double(1378.00));
        tm.put("Tod Hall", new Double(99.22));
        tm.put("Ralph Smith", new Double(-19.08));

        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = tm.entrySet();

        // Display the elements.
        for (Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // Deposit 1000 into John Doe's account.
        double balance = tm.get("John Doe");
        tm.put("John Doe", balance + 1000);

        System.out.println("John Doe's new balance: " +
            tm.get("John Doe"));
    }
}
```

The following is the output from this program:
Notice that TreeMap sorts the keys. However, in this case, they are sorted by first name instead of last name. You can alter this behavior by specifying a comparator when the map is created, as described shortly.

The LinkedHashMap Class

LinkedHashMap extends HashMap. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map.

That is, when iterating through a collection-view of a LinkedHashMap, the elements will be returned in the order in which they were inserted. You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed. LinkedHashMap is a generic class that has this declaration:

```java
class LinkedHashMap<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values.

LinkedHashMap defines the following constructors:

```java
LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int capacity)
```
LinkedHashMap(int capacity, float fillRatio)
LinkedHashMap(int capacity, float fillRatio, boolean Order)

The first form constructs a default LinkedHashMap. The second form initializes the LinkedHashMap with the elements from m. The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for HashMap. The default capacity is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If Order is true, then access order is used. If Order is false, then insertion order is used.

LinkedHashMap adds only one method to those defined by HashMap. This method is removeEldestEntry( ), and it is shown here:

protected boolean removeEldestEntry(Map.Entry<K, V> e)

This method is called by put( ) and putAll( ). The oldest entry is passed in e. By default, this method returns false and does nothing. However, if you override this method, then you can have the LinkedHashMap remove the oldest entry in the map. To do this, have your override return true. To keep the oldest entry, return false.

The IdentityHashMap Class

IdentityHashMap extends AbstractMap and implements
the Map interface. It is similar to HashMap except that it uses reference equality when comparing elements. IdentityHashMap is a generic class that has this declaration:

   class IdentityHashMap<K, V>

Here, K specifies the type of key, and V specifies the type of value. The API documentation explicitly states that IdentityHashMap is not for general use.

The EnumMap Class
EnumMap extends AbstractMap and implements Map. It is specifically for use with keys of an enum type. It is a generic class that has this declaration:

   class EnumMap<K extends Enum<K>, V>

Here, K specifies the type of key, and V specifies the type of value. Notice that K must extend Enum<K>, which enforces the requirement that the keys must be of an enum type.

   EnumMap defines the following constructors:

   EnumMap(Class<K> kType)
   EnumMap(Map<K, ? extends V> m)
   EnumMap(EnumMap<K, ? extends V> em)

The first constructor creates an empty EnumMap of type kType. The second creates an EnumMap map that contains the same entries as m. The third creates an EnumMap initialized with the values in em.
EnumMap defines no methods of its own.

Comparators
Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what “sorted order” means. By default, these classes store their elements by using what Java refers to as “natural ordering,” which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then specify a Comparator when you construct the set or map. Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.

Comparator is a generic interface that has this declaration:

```java
interface Comparator<T>
```

Here, T specifies the type of objects being compared.

```java
T h e Comparator interface defines two methods: compare( ) and equals( ). The compare( ) method, shown here, compares two elements for order:
```

```java
int compare(T obj1, T obj2)
```

obj1 and obj2 are the objects to be compared. Normally, this method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned. The method can throw a ClassCastException if the types of the objects are not compatible for comparison. By implementing
compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals( ) method, shown here, tests whether an object equals the invoking comparator:

```java
boolean equals(object obj)
```

Here, obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false. Overriding equals( ) is not necessary, and most simple comparators will not do so.

Using a Comparator

The following is an example that demonstrates the power of a custom comparator. It implements the compare( ) method for strings that operates in reverse of normal. Thus, it causes a tree set to be sorted in reverse order.
As the following output shows, the tree is now sorted in reverse order:

```
F E D C B A
```
Look closely at the MyComp class, which implements Comparator by implementing compare(). (As explained earlier, overriding equals() is neither necessary nor common.) Inside compare(), the String method compareTo() compares the two strings. However, bStr—not aStr—invokes compareTo(). This causes the outcome of the comparison to be reversed.

For a more practical example, the following program is an updated version of the TreeMap program shown earlier that stores account balances. In the previous version, the accounts were sorted by name, but the sorting began with the first name. The following program sorts the accounts by last name. To do so, it uses a comparator that compares the last name of each account. This results in the map being sorted by last name.

```java
// Use a comparator to sort accounts by last name.
import java.util.*;

// Compare last whole words in two strings.
class TComp implements Comparator<String> {
    public int compare(String a, String b) {
        int i, j, k;
        String aStr, bStr;

        aStr = a;
        bStr = b;

        // Find index of beginning of last name.
        i = aStr.lastIndexOf(' ');
        j = bStr.lastIndexOf(' ');

        k = aStr.substring(i).compareTo(bStr.substring(j));
        if(k==0) // last names match, check entire name
            return aStr.compareTo(bStr);
        else
            return k;
    }
}
```

Here is the output; notice that the accounts are now sorted by last name:

Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22

John Doe's new balance: 4434.34

The comparator class TComp compares two strings that hold first and last names. It does so by first comparing last names. To do this, it finds the index of
the last space in each string and then compares the substrings of each element that begin at that point. In cases where last names are equivalent, the first names are then compared. This yields a tree map that is sorted by last name, and within last name by first name. You can see this because Ralph Smith comes before Tom Smith in the output.

The Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class. They are summarized in Table 17-14. As explained earlier, beginning with JDK 5 all of the algorithms were retrofitted for generics.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static &lt;T&gt; boolean addAll(Collection &lt;? super T&gt; c, T... elements)</td>
<td>Inserts the elements specified by elements into the collection specified by c. Returns true if the elements were added and false otherwise.</td>
</tr>
<tr>
<td>static &lt;T&gt; Queue&lt;T&gt; asLifoQueue(Deque&lt;T&gt; c)</td>
<td>Returns a last-in, first-out view of c.</td>
</tr>
<tr>
<td>static &lt;T&gt; int binarySearch(List&lt;? extends T&gt; list, T value, Comparator&lt;? super T&gt; c)</td>
<td>Searches for value in list ordered according to c. Returns the position of value in list, or a negative value if value is not found.</td>
</tr>
<tr>
<td>static &lt;T&gt; int binarySearch(List&lt;? extends Comparable&lt;? super T&gt;&gt; list, T value)</td>
<td>Searches for value in list. The list must be sorted. Returns the position of value in list, or a negative value if value is not found.</td>
</tr>
<tr>
<td>static &lt;E&gt; Collection&lt;E&gt; checkedCollection(Collection&lt;E&gt; c, Class&lt;E&gt; t)</td>
<td>Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a ClassCastException.</td>
</tr>
<tr>
<td>Static Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>static &lt;E&gt; List&lt;E&gt; checkedList(List&lt;E&gt; c, Class&lt;E&gt; f)</code></td>
<td>Returns a run-time type-safe view of a <code>List</code>. An attempt to insert an incompatible element will cause a <code>ClassCastException</code>.</td>
</tr>
<tr>
<td>`static &lt;K, V&gt; Map&lt;K, V&gt; checkedMap(Map&lt;K, V&gt; c,</td>
<td>Returns a run-time type-safe view of a <code>Map</code>. An attempt to insert an incompatible element will cause a <code>ClassCastException</code>.</td>
</tr>
<tr>
<td>Class&lt;K&gt; keyT,</td>
<td>Class&lt;V&gt; valueT)</td>
</tr>
<tr>
<td><code>static &lt;E&gt; List&lt;E&gt; checkedSet(Set&lt;E&gt; c, Class&lt;E&gt; f)</code></td>
<td>Returns a run-time type-safe view of a <code>Set</code>. An attempt to insert an incompatible element will cause a <code>ClassCastException</code>.</td>
</tr>
<tr>
<td>`static &lt;K, V&gt; SortedMap&lt;K, V&gt; checkedSortedMap(SortedMap&lt;K, V&gt; c,</td>
<td>Returns a run-time type-safe view of a <code>SortedMap</code>. An attempt to insert an incompatible element will cause a <code>ClassCastException</code>.</td>
</tr>
<tr>
<td>Class&lt;K&gt; keyT,</td>
<td>Class&lt;V&gt; valueT)</td>
</tr>
<tr>
<td><code>static &lt;E&gt; SortedSet&lt;E&gt; checkedSortedSet(SortedSet&lt;E&gt; c, Class&lt;E&gt; f)</code></td>
<td>Returns a run-time type-safe view of a <code>SortedSet</code>. An attempt to insert an incompatible element will cause a <code>ClassCastException</code>.</td>
</tr>
<tr>
<td>`static &lt;T&gt; void copy(List&lt;? super T&gt; list1,</td>
<td>Copies the elements of <code>list2</code> to <code>list1</code>.</td>
</tr>
<tr>
<td>List&lt;? extends T&gt; list2)`</td>
<td></td>
</tr>
<tr>
<td>`static boolean disjoint(Collection&lt;?&gt; a,</td>
<td>Compares the elements in <code>a</code> to elements in <code>b</code>. Returns <code>true</code> if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns <code>false</code>.</td>
</tr>
<tr>
<td>Collection&lt;?&gt; b)`</td>
<td></td>
</tr>
<tr>
<td><code>static &lt;T&gt; Enumeration&lt;T&gt; emptyEnumeration()</code></td>
<td>Returns an empty enumeration, which is an enumeration with no elements. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>static &lt;T&gt; Iterator&lt;T&gt; emptyIterator()</code></td>
<td>Returns an empty iterator, which is an iterator with no elements. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>static &lt;T&gt; List&lt;T&gt; emptyList()</code></td>
<td>Returns an immutable, empty <code>List</code> object of the inferred type.</td>
</tr>
<tr>
<td><code>static &lt;T&gt; ListIterator&lt;T&gt; emptyListIterator()</code></td>
<td>Returns an empty list iterator, which is a list iterator that has no elements. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>static &lt;K, V&gt; Map&lt;K, V&gt; emptyMap()</code></td>
<td>Returns an immutable, empty <code>Map</code> object of the inferred type.</td>
</tr>
<tr>
<td><code>static &lt;T&gt; Set&lt;T&gt; emptySet()</code></td>
<td>Returns an immutable, empty <code>Set</code> object of the inferred type.</td>
</tr>
<tr>
<td><code>static &lt;T&gt; Enumeration&lt;T&gt; enumeration(Collection&lt;T&gt; c)</code></td>
<td>Returns an enumeration over <code>c</code>. (See “The Enumeration Interface,” later in this chapter.)</td>
</tr>
<tr>
<td><code>static &lt;T&gt; void fill(List&lt;? super T&gt; list, T obj)</code></td>
<td>Assigns <code>obj</code> to each element of <code>list</code>.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>static int frequency(Collection&lt;? extends T&gt; c, object obj)</td>
<td>Counts the number of occurrences of obj in c and returns the result.</td>
</tr>
<tr>
<td>static int indexOfSubList(List&lt;? extends T&gt; list, List&lt;? extends T&gt; subList)</td>
<td>Searches list for the first occurrence of subList. Returns the index of the first match, or -1 if no match is found.</td>
</tr>
<tr>
<td>static int lastIndexOfSubList(List&lt;? extends T&gt; list, List&lt;? extends T&gt; subList)</td>
<td>Searches list for the last occurrence of subList. Returns the index of the last match, or -1 if no match is found.</td>
</tr>
<tr>
<td>static &lt;T&gt; ArrayList&lt;T&gt; list(Enumeration&lt;T&gt; enum)</td>
<td>Returns an ArrayList that contains the elements of enum.</td>
</tr>
<tr>
<td>static &lt;T extends Object &amp; Comparable&lt;? super T&gt;&gt; T max(Collection&lt;? extends T&gt; c)</td>
<td>Returns the maximum element in c as determined by comp.</td>
</tr>
<tr>
<td>static &lt;T extends Object &amp; Comparable&lt;? super T&gt;&gt; T min(Collection&lt;? extends T&gt; c)</td>
<td>Returns the minimum element in c as determined by natural ordering. The collection need not be sorted.</td>
</tr>
<tr>
<td>static &lt;T&gt; List&lt;T&gt; nCopies(int num, T obj)</td>
<td>Returns num copies of obj contained in an immutable list. num must be greater than or equal to zero.</td>
</tr>
<tr>
<td>static &lt;E&gt; Set&lt;E&gt; newSetFromMap(Map&lt;E, Boolean&gt; m)</td>
<td>Creates and returns a set backed by the map specified by m, which must be empty at the time this method is called.</td>
</tr>
<tr>
<td>static &lt;T&gt; boolean replaceAll(List&lt;T&gt; list, T old, T new)</td>
<td>Replaces all occurrences of old with new in list. Returns true if at least one replacement occurred. Returns false, otherwise.</td>
</tr>
<tr>
<td>static void reverse(List&lt;T&gt; list)</td>
<td>Reverses the sequence in list.</td>
</tr>
<tr>
<td>static &lt;T&gt; Comparator&lt;T&gt; reverseOrder(Comparator&lt;T&gt; comp)</td>
<td>Returns a reverse comparator based on the one passed in comp. That is, the returned comparator reverses the outcome of a comparison that uses comp.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>static Comparator(&lt;T&gt;) reverseOrder()</td>
<td>Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements.</td>
</tr>
<tr>
<td>static void rotate(List(&lt;T&gt;) list, int n)</td>
<td>Rotates list by n places to the right. To rotate left, use a negative value for n.</td>
</tr>
<tr>
<td>static void shuffle(List(&lt;T&gt;) list, Random r)</td>
<td>Shuffles (i.e., randomizes) the elements in list by using r as a source of random numbers.</td>
</tr>
<tr>
<td>static void shuffle(List(&lt;T&gt;) list)</td>
<td>Shuffles (i.e., randomizes) the elements in list.</td>
</tr>
<tr>
<td>static Set(&lt;T&gt;) singleton(T obj)</td>
<td>Returns obj as an immutable set. This is an easy way to convert a single object into a set.</td>
</tr>
<tr>
<td>static List(&lt;T&gt;) singletonList(T obj)</td>
<td>Returns obj as an immutable list. This is an easy way to convert a single object into a list.</td>
</tr>
<tr>
<td>static Map(K, V) singletonMap(K k, V v)</td>
<td>Returns the key/value pair k/v as an immutable map. This is an easy way to convert a single key/value pair into a map.</td>
</tr>
<tr>
<td>static void sort(List(&lt;T&gt;) list, Comparator(&lt;?\ super T&gt;\ comp)</td>
<td>Sorts the elements of list as determined by comp.</td>
</tr>
<tr>
<td>static T extends Comparable(&lt;?\ super T&gt;&gt;&gt; void sort(List(&lt;T&gt;) list)</td>
<td>Sorts the elements of list as determined by their natural ordering.</td>
</tr>
<tr>
<td>static void swap(List&lt;?&gt; list, int idx1, int idx2)</td>
<td>Exchanges the elements in list at the indices specified by idx1 and idx2.</td>
</tr>
<tr>
<td>static Collection(&lt;T&gt;) synchronizedCollection(Collection(&lt;T&gt;) c)</td>
<td>Returns a thread-safe collection backed by c.</td>
</tr>
<tr>
<td>static List(&lt;T&gt;) synchronizedList(List(&lt;T&gt;) list)</td>
<td>Returns a thread-safe list backed by list.</td>
</tr>
<tr>
<td>static Map(K, V) synchronizedMap(Map(&lt;K, V) m)</td>
<td>Returns a thread-safe map backed by m.</td>
</tr>
<tr>
<td>static Set(&lt;T&gt;) synchronizedSet(Set(&lt;T&gt;) s)</td>
<td>Returns a thread-safe set backed by s.</td>
</tr>
<tr>
<td>static SortedMap(K, V) synchronizedSortedMap(SortedMap(&lt;K, V&gt; sm)</td>
<td>Returns a thread-safe sorted map backed by sm.</td>
</tr>
<tr>
<td>static SortedSet(&lt;T&gt;) synchronizedSortedSet(SortedSet(&lt;T&gt;) ss)</td>
<td>Returns a thread-safe sorted set backed by ss.</td>
</tr>
</tbody>
</table>
Table 17-14 The Algorithms Defined by Collections

Several of the methods can throw a ClassCastException, which occurs when an attempt is made to compare incompatible types, or an UnsupportedOperationException, which occurs when an attempt is made to modify an unmodifiable collection. Other exceptions are possible, depending on the method.

One thing to pay special attention to is the set of checked methods, such as checkedCollection( ), which returns what the API documentation refers to as a “dynamically typesafe view” of a collection. This view is a reference to the collection that monitors insertions into the collection for type compatibility at run time. An attempt to insert an incompatible element will cause a ClassCastException. Using such a view is especially helpful during debugging because it ensures that the collection always contains valid elements. Related methods include checkedSet( ), checkedList( ), checkedMap( ), and so on. They obtain a type-safe view.
for the indicated collection.

Notice that several methods, such as synchronizedList() and synchronizedSet(), are used to obtain synchronized (thread-safe) copies of the various collections. As a general rule, the standard collections implementations are not synchronized. You must use the synchronization algorithms to provide synchronization. One other point: iterators to synchronized collections must be used within synchronized blocks.

The set of methods that begins with unmodifiable returns views of the various collections that cannot be modified. These will be useful when you want to grant some process read—but not write—capabilities on a collection.

Collections defines three static variables: EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. All are immutable.

The following program demonstrates some of the algorithms. It creates and initializes a linked list. The reverseOrder() method returns a Comparator that reverses the comparison of Integer objects. The list elements are sorted according to this comparator and then are displayed. Next, the list is randomized by calling shuffle(), and then its minimum and maximum values are displayed.

```java
import java.util.*;

class AlgorithmsDemo {
    public static void main(String args[]) {

        // Create and initialize linked list.
        LinkedList<Integer> ll = new LinkedList<Integer>();

        // Demonstrate various algorithms.
```
ll.add(-8);
ll.add(20);
ll.add(-20);
ll.add(8);

// Create a reverse order comparator.
Comparator<Integer> r = Collections.reverseOrder();

// Sort list by using the comparator.
Collections.sort(ll, r);

System.out.print("List sorted in reverse: ");
for(int i : ll)
    System.out.print(i+ " ");

System.out.println();

// Shuffle list.
Collections.shuffle(ll);

// Display randomized list.
System.out.print("List shuffled: ");
for(int i : ll)
    System.out.print(i + " ");

System.out.println();
System.out.println("Minimum: " + Collections.min(ll));
System.out.println("Maximum: " + Collections.max(ll));
}

Output from this program is shown here:

List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20

Notice that min( ) and max( ) operate on the list after it has been shuffled. Neither requires a sorted list for its operation.

Arrays
The Arrays class provides various methods that are useful
when working with arrays. These methods help bridge the gap between collections and arrays. Each method defined by Arrays is examined in this section.

The asList( ) method returns a List that is backed by a specified array. In other words, both the list and the array refer to the same location. It has the following signature:

```java
static <T> List asList(T... array)
```

Here, array is the array that contains the data.

The binarySearch( ) method uses a binary search to find a specified value. This method must be applied to sorted arrays. Here are some of its forms. (Additional forms let you search a subrange):

```java
static int binarySearch(byte array[], byte value)
static int binarySearch(char array[], char value)
static int binarySearch(double array[], double value)
static int binarySearch(float array[], float value)
static int binarySearch(int array[], int value)
static int binarySearch(long array[], long value)
static int binarySearch(short array[], short value)
static int binarySearch(Object array[], Object value)
static <T> int binarySearch(T[] array, T value, Comparator<? super T> c)
```

Here, array is the array to be searched, and value is the value to be located. The last two forms throw a ClassCastException if array contains elements that cannot be compared (for example, Double and StringBuffer) or
if value is not compatible with the types in array. In the last form, the Comparator c is used to determine the order of the elements in array. In all cases, if value exists in array, the index of the element is returned. Otherwise, a negative value is returned.

The copyOf( ) method returns a copy of an array and has the following forms:

```
static boolean[ ] copyOf(boolean[ ] source, int len)
static byte[ ] copyOf(byte[ ] source, int len)
static char[ ] copyOf(char[ ] source, int len)
static double[ ] copyOf(double[ ] source, int len)
static float[ ] copyOf(float[ ] source, int len)
static int[ ] copyOf(int[ ] source, int len)
static long[ ] copyOf(long[ ] source, int len)
static short[ ] copyOf(short[ ] source, int len)
static <T> T[ ] copyOf(T[ ] source, int len)
static <T,U> T[ ] copyOf(U[ ] source, int len,
   Class<? extends T[ ]> resultT)
```

The original array is specified by source, and the length of the copy is specified by len. If the copy is longer than source, then the copy is padded with zeros (for numeric arrays), nulls (for object arrays), or false (for boolean arrays). If the copy is shorter than source, then the copy is truncated. In the last form, the type of resultT becomes the type of the array returned. If len is negative, a NegativeArraySizeException is thrown. If source is null, a NullPointerException is thrown. If resultT is incompatible with the type of source, an
ArrayStoreException is thrown.

The copyOfRange( ) method returns a copy of a range within an array and has the following forms:

```java
static boolean[ ] copyOfRange(boolean[ ] source, int start, int end)
static byte[ ] copyOfRange(byte[ ] source, int start, int end)
static char[ ] copyOfRange(char[ ] source, int start, int end)
static double[ ] copyOfRange(double[ ] source, int start, int end)
static float[ ] copyOfRange(float[ ] source, int start, int end)
static int[ ] copyOfRange(int[ ] source, int start, int end)
static long[ ] copyOfRange(long[ ] source, int start, int end)
static short[ ] copyOfRange(short[ ] source, int start, int end)
static <T> T[ ] copyOfRange(T[ ] source, int start, int end)
static <T,U> T[ ] copyOfRange(U[ ] source, int start, int end,
                                Class<? extends T[ ]> resultT)
```

The original array is specified by source. The range to copy is specified by the indices passed via start and end. The range runs from start to end – 1. If the range is
longer than source, then the copy is padded with zeros (for numeric arrays), nulls (for object arrays), or false (for boolean arrays). In the last form, the type of resultT becomes the type of the array returned. If start is negative or greater than the length of source, an ArrayIndexOutOfBoundsException is thrown. If start is greater than end, an IllegalArgumentException is thrown. If source is null, a NullPointerException is thrown. If resultT is incompatible with the type of source, an ArrayStoreException is thrown.

The equals() method returns true if two arrays are equivalent. Otherwise, it returns false. The equals() method has the following forms:

static boolean equals(boolean array1[ ], boolean array2 [ ])
static boolean equals(byte array1[ ], byte array2 [ ])
static boolean equals(char array1[ ], char array2 [ ])
static boolean equals(double array1[ ], double array2 [ ])
static boolean equals(float array1[ ], float array2 [ ])
static boolean equals(int array1[ ], int array2 [ ])
static boolean equals(long array1[ ], long array2 [ ])
static boolean equals(short array1[ ], short array2 [ ])
static boolean equals(Object array1[ ], Object array2 [ ])

Here, array1 and array2 are the two arrays that are compared for equality.

The deepEquals() method can be used to determine if
two arrays, which might contain nested arrays, are equal. It has this declaration:

```java
static boolean deepEquals(Object[] a, Object[] b)
```

It returns true if the arrays passed in `a` and `b` contain the same elements. If `a` and `b` contain nested arrays, then the contents of those nested arrays are also checked. It returns false if the arrays, or any nested arrays, differ.

The `fill()` method assigns a value to all elements in an array. In other words, it fills an array with a specified value. The `fill()` method has two versions. The first version, which has the following forms, fills an entire array:

```java
static void fill(boolean array[], boolean value)
static void fill(byte array[], byte value)
static void fill(char array[], char value)
static void fill(double array[], double value)
static void fill(float array[], float value)
static void fill(int array[], int value)
static void fill(long array[], long value)
static void fill(short array[], short value)
static void fill(Object array[], Object value)
```

Here, value is assigned to all elements in `array`.

The second version of the `fill()` method assigns a value to a subset of an array. Its forms are shown here:

```java
static void fill(boolean array[], int start, int end, boolean value)
```
static void fill(byte array[], int start, int end, byte value)
static void fill(char array[], int start, int end, char value)
static void fill(double array[], int start, int end, double value)
static void fill(float array[], int start, int end, float value)
static void fill(int array[], int start, int end, int value)
static void fill(long array[], int start, int end, long value)
static void fill(short array[], int start, int end, short value)
static void fill(Object array[], int start, int end, Object value)

Here, value is assigned to the elements in array from position start to position end–1. These methods may all throw an IllegalArgumentException if start is greater than end, or an ArrayIndexOutOfBoundsException if start or end is out of bounds. An ArrayStoreException is possible with the Object versions.

The sort() method sorts an array so that it is arranged in ascending order. The sort() method has two versions. The first version, shown here, sorts the entire array:

    static void sort(byte array[])
    static void sort(char array[])
    static void sort(double array[])
    static void sort(float array[])
    static void sort(int array[])
    static void sort(long array[])
    static void sort(short array[])

    static void sort(Object array[])
Here, array is the array to be sorted. In the last form, c is a Comparator that is used to order the elements of array. The last two forms can throw a ClassCastException if elements of the array being sorted are not comparable.

The second version of sort() enables you to specify a range within an array that you want to sort. Its forms are shown here:

```java
static void sort(byte array[], int start, int end)
static void sort(char array[], int start, int end)
static void sort(double array[], int start, int end)
static void sort(float array[], int start, int end)
static void sort(int array[], int start, int end)
static void sort(long array[], int start, int end)
static void sort(short array[], int start, int end)
static void sort(Object array[], int start, int end)
static <T> void sort(T array[], int start, int end,
                     Comparator<? super T> c)
```

Here, the range beginning at start and running through end−1 within array will be sorted. In the last form, c is a Comparator that is used to order the elements of array. All of these methods can throw an IllegalArgumentException if start is greater than end, or
an ArrayIndexOutOfBoundsException if start or end is out of bounds. The last two forms can also throw a ClassCastException if elements of the array being sorted are not comparable.

Arrays also provides toString() and hashCode() for the various types of arrays. In addition, deepToString() and deepHashCode() are provided, which operate effectively on arrays that contain nested arrays.

The following program illustrates how to use some of the methods of the Arrays class:

```java
// Demonstrate Arrays
import java.util.*;

class ArraysDemo {
    public static void main(String args[]) {

        // Allocate and initialize array.
        int array[] = new int[10];
        for(int i = 0; i < 10; i++)
            array[i] = -3 * i;

        // Display, sort, and display the array.
        System.out.print("Original contents: ");
        display(array);
        Arrays.sort(array);
        System.out.print("Sorted: ");
        display(array);

        // Fill and display the array.
        Arrays.fill(array, 2, 6, -1);
        System.out.print("After fill(): ");
        display(array);
    }
}
```
The following is the output from this program:

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```

Why Generic Collections?
As mentioned at the start of this chapter, the entire Collections Framework was refitted for generics when JDK 5 was released. Furthermore, the Collections Framework is arguably the single most important use of generics in the Java API. The reason for this is that generics add type safety to the Collections Framework. Before moving on, it is worth taking some time to examine in detail the significance of this improvement. It will also show why older pre-generics collection-based
Let's begin with an example that uses pre-generics code. The following program stores a list of strings in an ArrayList and then displays the contents of the list:

```java
// Pre-generics example that uses a collection.
import java.util.*;

class OldStyle {
    public static void main(String args[]) {
        ArrayList list = new ArrayList();

        // These lines store strings, but any type of object can be stored. In old-style code, there is no convenient way to restrict the type of objects stored in a collection
        list.add("one");
        list.add("two");
        list.add("three");
        list.add("four");

        Iterator itr = list.iterator();
        while(itr.hasNext()) {
            // To retrieve an element, an explicit type cast is needed because the collection stores only Object.
            String str = (String) itr.next(); // explicit cast needed here.

            System.out.println(str + " is " + str.length() + " chars long.");
        }
    }
}
```

Prior to generics, all collections stored references of type Object. This allowed any type of reference to be stored in the collection. The preceding program uses this feature to store references to objects of type String in list, but any type of reference could have been stored.

Unfortunately, the fact that a pre-generics collection stored Object references could easily lead to errors. First, it required that you, rather than the compiler, ensure that
only objects of the proper type be stored in a specific collection. For example, in the preceding example, list is clearly intended to store Strings, but there is nothing that actually prevents another type of reference from being added to the collection. For example, the compiler will find nothing wrong with this line of code:

```java
list.add(new Integer(100));
```

Because list stores Object references, it can store a reference to Integer as well as it can store a reference to String. However, if you intended list to hold only strings, then the preceding statement would corrupt the collection. Again, the compiler had no way to know that the preceding statement is invalid.

The second problem with pre-generics collections is that when you retrieve a reference from the collection, you must manually cast that reference into the proper type. This is why the preceding program casts the reference returned by `next()` into String. Prior to generics, collections simply stored Object references. Thus, the cast was necessary when retrieving objects from a collection.

Aside from the inconvenience of always having to cast a retrieved reference into its proper type, this lack of type safety often led to a rather serious, but surprisingly easy-to-create, error. Because Object can be cast into any type of object, it was possible to cast a reference obtained from a collection into the wrong type. For example, if the following statement were added to the preceding example, it would still compile without error,
but generate a run-time exception when executed:

```java
Integer i = (Integer) itr.next();
```

Recall that the preceding example stored only references to instances of type `String` in list. Thus, when this statement attempts to cast a `String` into an `Integer`, an invalid cast exception results! Because this happens at run time, this is a very serious error.

The addition of generics fundamentally improves the usability and safety of collections because it

- Ensures that only references to objects of the proper type can actually be stored in a collection. Thus, a collection will always contain references of a known type.
- Eliminates the need to cast a reference retrieved from a collection. Instead, a reference retrieved from a collection is automatically cast into the proper type. This prevents run-time errors due to invalid casts and avoids an entire category of errors.

These two improvements are made possible because each collection class has been given a type parameter that specifies the type of the collection. For example, `ArrayList` is now declared like this:

```java
class ArrayList<E>
```

Here, `E` is the type of element stored in the collection. Therefore, the following declares an `ArrayList` for objects of type `String`:

```java
ArrayList<String> list = new ArrayList<String>();
```
Now, only references of type String can be added to list.

The Iterator and ListIterator interfaces are now also generic. This means that the type parameter must agree with the type of the collection for which the iterator is obtained. Furthermore, this type compatibility is enforced at compile time.

The following program shows the modern, generic form of the preceding program:

```java
// Modern, generics version.
import java.util.*;

class NewStyle {
    public static void main(String args[]) {
        // Now, list holds references of type String.
        ArrayList<String> list = new ArrayList<String>();
        list.add("one");
        list.add("two");
        list.add("three");
        list.add("four");

        // Notice that Iterator is also generic.
        Iterator<String> itr = list.iterator();

        // The following statement will now cause a compile-time error.
        //    Iterator<Integer> itr = list.iterator(); // Error!

        while(itr.hasNext()) {
            String str = itr.next(); // no cast needed
            System.out.println(str + " is " + str.length() + " chars long.");
        }
    }
}
```

Now, list can hold only references to objects of type String.
Furthermore, as the following line shows, there is no need to cast the return value of `next()` into `String`:

```java
String str = itr.next(); // no cast needed
```

The cast is performed automatically.

Because of support for raw types, older, pre-generics collection code will continue to compile and run. However, all new code should use generics, and you should update older code as soon as time permits. The addition of generics to the Collections Framework was a fundamental improvement that should be utilized wherever possible.

The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of `java.util` did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects. When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces. Thus, they are now technically part of the Collections Framework. However, where a modern collection duplicates the functionality of a legacy class, you will usually want to use the newer collection class. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the modern collection classes described in this chapter are synchronized, but all the legacy classes are synchronized. This distinction may be important in some situations. Of course, you can easily
synchronize collections by using one of the algorithms provided by Collections.

The legacy classes defined by java.util are shown here:

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Hashtable</th>
<th>Properties</th>
<th>Stack</th>
<th>Vector</th>
</tr>
</thead>
</table>

There is one legacy interface called Enumeration. The following sections examine Enumeration and each of the legacy classes, in turn.

The Enumeration Interface

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as Vector and Properties), is used by several other API classes, and is currently in widespread use in application code. Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:

```java
interface Enumeration<E>
```

where E specifies the type of element being enumerated.

Enumeration specifies the following two methods:

```java
boolean hasMoreElements( )
E nextElement( )
```

When implemented, `hasMoreElements( )` must return true while there are still more elements to extract, and false when all the elements have been enumerated.
nextElement( ) returns the next object in the enumeration. That is, each call to nextElement( ) obtains the next object in the enumeration. It throws NoSuchElementException when the enumeration is complete.

Vector

Vector implements a dynamic array. It is similar to ArrayList, but with two differences: Vector is synchronized, and it contains many legacy methods that duplicate the functionality of methods defined by the Collections Framework. With the advent of collections, Vector was reengineered to extend AbstractList and to implement the List interface. With the release of JDK 5, it was retrofitted for generics and reengineered to implement Iterable. This means that Vector is fully compatible with collections, and a Vector can have its contents iterated by the enhanced for loop.

Vector is declared like this:

class Vector<E>

Here, E specifies the type of element that will be stored.

Here are the Vector constructors:

Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)

The first form creates a default vector, which has an
initial size of 10. The second form creates a vector whose initial capacity is specified by size. The third form creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward. The fourth form creates a vector that contains the elements of collection c.

All vectors start with an initial capacity. After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place as the vector grows. This reduction is important, because allocations are costly in terms of time. The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If you don’t specify an increment, the vector’s size is doubled by each allocation cycle.

Vector defines these protected data members:

```c
int capacityIncrement;
int elementCount;
Object[ ] elementData;
```

The increment value is stored in capacityIncrement. The number of elements currently in the vector is stored in elementCount. The array that holds the vector is stored in elementData.
In addition to the collections methods defined by List, Vector defines several legacy methods, which are summarized in Table 17-15.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addElement(E element)</td>
<td>The object specified by element is added to the vector.</td>
</tr>
<tr>
<td>int capacity( )</td>
<td>Returns the capacity of the vector.</td>
</tr>
<tr>
<td>Object clone( )</td>
<td>Returns a duplicate of the invoking vector.</td>
</tr>
<tr>
<td>boolean contains(Object element)</td>
<td>Returns true if element is contained by the vector, and returns false if it is not.</td>
</tr>
<tr>
<td>void copyInto(Object array[ ])</td>
<td>The elements contained in the invoking vector are copied into the array specified by array.</td>
</tr>
<tr>
<td>E elementAt(int index)</td>
<td>Returns the element at the location specified by index.</td>
</tr>
<tr>
<td>Enumeration&lt;E&gt; elements( )</td>
<td>Returns an enumeration of the elements in the vector.</td>
</tr>
<tr>
<td>void ensureCapacity(int size)</td>
<td>Sets the minimum capacity of the vector to size.</td>
</tr>
<tr>
<td>E firstElement( )</td>
<td>Returns the first element in the vector.</td>
</tr>
<tr>
<td>int indexOf(Object element)</td>
<td>Returns the index of the first occurrence of element. If the object is not in the vector, -1 is returned.</td>
</tr>
<tr>
<td>int indexOf(Object element, int start)</td>
<td>Returns the index of the first occurrence of element at or after start. If the object is not in that portion of the vector, -1 is returned.</td>
</tr>
<tr>
<td>void insertElementAt(E element, int index)</td>
<td>Adds element to the vector at the location specified by index.</td>
</tr>
<tr>
<td>boolean isEmpty( )</td>
<td>Returns true if the vector is empty, and returns false if it contains one or more elements.</td>
</tr>
</tbody>
</table>
Table 17-15 The Legacy Methods Defined by Vector

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E lastElement( )</td>
<td>Returns the last element in the vector.</td>
</tr>
<tr>
<td>int lastIndexOf(Object element)</td>
<td>Returns the index of the last occurrence of element. If the object is not in</td>
</tr>
<tr>
<td></td>
<td>the vector, -1 is returned.</td>
</tr>
<tr>
<td>int lastIndexOf(Object element, int start)</td>
<td>Returns the index of the last occurrence of element before start. If the</td>
</tr>
<tr>
<td></td>
<td>object is not in that portion of the vector, -1 is returned.</td>
</tr>
<tr>
<td>void removeAllElements( )</td>
<td>Empties the vector. After this method executes, the size of the vector is</td>
</tr>
<tr>
<td></td>
<td>zero.</td>
</tr>
<tr>
<td>boolean removeElement(Object element)</td>
<td>Removes element from the vector. If more than one instance of the specified</td>
</tr>
<tr>
<td></td>
<td>object exists in the vector, then it is the first one that is removed.</td>
</tr>
<tr>
<td></td>
<td>Returns true if successful and false if the object is not found.</td>
</tr>
<tr>
<td>void removeElementAt(int index)</td>
<td>Removes the element at the location specified by index.</td>
</tr>
<tr>
<td>void setElementAt(E element, int index)</td>
<td>The location specified by index is assigned element.</td>
</tr>
<tr>
<td>void setSize(int size)</td>
<td>Sets the number of elements in the vector to size. If the new size is less</td>
</tr>
<tr>
<td></td>
<td>than the old size, elements are lost. If the new size is larger than the</td>
</tr>
<tr>
<td></td>
<td>old size, null elements are added.</td>
</tr>
<tr>
<td>int size( )</td>
<td>Returns the number of elements currently in the vector.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns the string equivalent of the vector.</td>
</tr>
<tr>
<td>void trimToSize( )</td>
<td>Sets the vector’s capacity equal to the number of elements that it currently</td>
</tr>
<tr>
<td></td>
<td>holds.</td>
</tr>
</tbody>
</table>

Because Vector implements List, you can use a vector just like you use an ArrayList instance. You can also manipulate one using its legacy methods. For example, after you instantiate a Vector, you can add an element to it by calling addElement( ). To obtain the element at a specific location, call elementAt( ). To obtain the first element in the vector, call firstElement( ). To retrieve the last element, call lastElement( ). You can obtain the index of an element by using indexOf( ) and lastIndexOf( ). To remove an element, call removeElement( ) or removeElementAt( ).

The following program uses a vector to store various types of numeric objects. It demonstrates several of the
legacy methods defined by Vector. It also demonstrates
the Enumeration interface.

```
// Demonstrate various Vector operations.
import java.util.*;

class VectorDemo {
    public static void main(String args[]) {

        // initial size is 3, increment is 2
        Vector<Integer> v = new Vector<Integer>(3, 2);

        System.out.println("Initial size: "+ v.size());
        System.out.println("Initial capacity: "+ v.capacity());

        v.addElement(1);
        v.addElement(2);
        v.addElement(3);
        v.addElement(4);

        System.out.println("Capacity after four additions: " +
                          v.capacity());

        v.addElement(5);
        System.out.println("Current capacity: "+ v.capacity());

        v.addElement(6);
        v.addElement(7);

        System.out.println("Current capacity: " +
                          v.capacity());

        v.addElement(9);
        v.addElement(10);

        System.out.println("Current capacity: " +
                          v.capacity());
    }
}
The output from this program is shown here:

Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5 6 7 9 10 11 12

Instead of relying on an enumeration to cycle through the objects (as the preceding program does), you can use an iterator. For example, the following iterator-based code can be substituted into the program:

```java
// Use an iterator to display contents.
Iterator<Integer> vItr = v.iterator();

System.out.println("\nElements in vector:");
while(vItr.hasNext())
    System.out.print(vItr.nextElement() + " ");
System.out.println();
```
You can also use a for-each for loop to cycle through a Vector, as the following version of the preceding code shows:

```java
// Use an enhanced for loop to display contents
System.out.println("\nElements in vector:");
for(int i : v)
    System.out.print(i + " ");
System.out.println();
```

Because the Enumeration interface is not recommended for new code, you will usually use an iterator or a for-each for loop to enumerate the contents of a vector. Of course, much legacy code exists that employs Enumeration. Fortunately, enumerations and iterators work in nearly the same manner.

Stack

Stack is a subclass of Vector that implements a standard last-in, first-out stack. Stack only defines the default constructor, which creates an empty stack. With the release of JDK 5, Stack was retrofitted for generics and is declared as shown here:

```java
class Stack<E>
```

Here, E specifies the type of element stored in the stack.

Stack includes all the methods defined by Vector and adds several of its own, shown in Table 17-16.

To put an object on the top of the stack, call push().
To remove and return the top element, call `pop( )`. You can use `peek( )` to return, but not remove, the top object. An `EmptyStackException` is thrown if you call `pop( )` or `peek( )` when the invoking stack is empty. The `empty( )` method returns `true` if nothing is on the stack. The `search( )` method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several `Integer` objects onto it, and then pops them off again:

```java
// Demonstrate the Stack class.
import java.util.*;

class StackDemo {
    static void showpush(Stack<Integer> st, int a) {
        st.push(a);
        System.out.println("push( + a + ")");
        System.out.println("stack: " + st);
    }

    static void showpop(Stack<Integer> st) {
        System.out.print("pop -> ");
        Integer a = st.pop();
        System.out.println(a);
        System.out.println("stack: " + st);
    }

    public static void main(String args[]) {
        Stack<Integer> st = new Stack<Integer>();
    }
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean empty( )</td>
<td>Returns <code>true</code> if the stack is empty, and returns <code>false</code> if the stack contains elements.</td>
</tr>
<tr>
<td>E peek( )</td>
<td>Returns the element on the top of the stack, but does not remove it.</td>
</tr>
<tr>
<td>E pop( )</td>
<td>Returns the element on the top of the stack, removing it in the process.</td>
</tr>
<tr>
<td>E push(E element)</td>
<td>Pushes <code>element</code> onto the stack. <code>element</code> is also returned.</td>
</tr>
<tr>
<td>int search(Object element)</td>
<td>Searches for <code>element</code> in the stack. If found, its offset from the top of the stack is returned. Otherwise, −1 is returned.</td>
</tr>
</tbody>
</table>

**Table 17-16 The Methods Defined by Stack**
System.out.println("stack: "+ st);
showpush(st, 42);
showpush(st, 66);
showpush(st, 99);
showpop(st);
showpop(st);
showpop(st);

try {
    showpop(st);
} catch (EmptyStackException e) {
    System.out.println("empty stack");
}

}
}

The following is the output produced by the program; notice how the exception handler for EmptyStackException is caught so that you can gracefully handle a stack underflow:

stack: []
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: []
pop -> empty stack

One other point: Although Stack is not deprecated, ArrayDeque is a better choice.

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. Given a
key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Although not currently deprecated, Dictionary is classified as obsolete, because it is fully superseded by Map. However, Dictionary is still in use and thus is fully discussed here.

With the advent of JDK 5, Dictionary was made generic. It is declared as shown here:

```java
class Dictionary<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values. The abstract methods defined by Dictionary are listed in Table 17-17.

To add a key and a value, use the put() method. Use get() to retrieve the value of a given key. The keys and values can each be returned as an Enumeration by the keys() and elements() methods, respectively. The size() method returns the number of key/value pairs stored in a dictionary, and isEmpty() returns true when the dictionary is empty. You can use the remove() method to delete a key/value pair.
Table 17-17 The Abstract Methods Defined by Dictionary

REMEMBER

The Dictionary class is obsolete. You should implement the Map interface to obtain key/value storage functionality.

Hashtable

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. However, with the advent of collections, Hashtable was reengineered to also implement the Map interface. Thus, Hashtable is integrated into the Collections Framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. However, neither keys nor values can be null. When using a Hashtable, you specify an object that is
used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Hashtable was made generic by JDK 5. It is declared like this:

```java
class Hashtable<K, V>
```

Here, K specifies the type of keys, and V specifies the type of values.

A hash table can only store objects that override the `hashCode()` and `equals()` methods that are defined by `Object`. The `hashCode()` method must compute and return the hash code for the object. Of course, `equals()` compares two objects. Fortunately, many of Java’s built-in classes already implement the `hashCode()` method. For example, the most common type of `Hashtable` uses a `String` object as the key. `String` implements both `hashCode()` and `equals()`.

The `Hashtable` constructors are shown here:

```java
Hashtable( )
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map<? extends K, ? extends V> m)
```

The first version is the default constructor. The second version creates a hash table that has an initial size specified by `size`. (The default size is 11.) The third version creates a hash table that has an initial size
specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used. Finally, the fourth version creates a hash table that is initialized with the elements in m. The default load factor of 0.75 is used.

In addition to the methods defined by the Map interface, which Hashtable now implements, Hashtable defines the legacy methods listed in Table 17-18. Several methods throw NullPointerException if an attempt is made to use a null key or value.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear( )</td>
<td>Resets and empties the hash table.</td>
</tr>
<tr>
<td>Object clone( )</td>
<td>Returns a duplicate of the invoking object.</td>
</tr>
<tr>
<td>boolean contains(Object value)</td>
<td>Returns true if some value equal to value exists within the hash table. Returns false if the value isn’t found.</td>
</tr>
<tr>
<td>boolean containsKey(Object key)</td>
<td>Returns true if some key equal to key exists within the hash table. Returns false if the key isn’t found.</td>
</tr>
<tr>
<td>boolean containsValue(Object value)</td>
<td>Returns true if some value equal to value exists within the hash table. Returns false if the value isn’t found.</td>
</tr>
<tr>
<td>Enumeration&lt;V&gt; elements( )</td>
<td>Returns an enumeration of the values contained in the hash table.</td>
</tr>
<tr>
<td>V get(Object key)</td>
<td>Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Returns true if the hash table is empty; returns false if it contains at least one key.</td>
</tr>
<tr>
<td>Enumeration&lt; K &gt; keys()</td>
<td>Returns an enumeration of the keys contained in the hash table.</td>
</tr>
<tr>
<td>V put(K key, V value)</td>
<td>Inserts a key and a value into the hash table. Returns null if key isn’t already in the hash table; returns the previous value associated with key if key is already in the hash table.</td>
</tr>
<tr>
<td>void rehash()</td>
<td>Increases the size of the hash table and rehashes all of its keys.</td>
</tr>
<tr>
<td>V remove(Object key)</td>
<td>Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned.</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the number of entries in the hash table.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns the string equivalent of a hash table.</td>
</tr>
</tbody>
</table>

Table 17-18 The Legacy Methods Defined by Hashtable

The following example reworks the bank account program, shown earlier, so that it uses a Hashtable to store the names of bank depositors and their current balances:
The output from this program is shown here:

Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

John Doe's new balance: 4434.34
One important point: Like the map classes, Hashtable does not directly support iterators. Thus, the preceding program uses an enumeration to display the contents of balance. However, you can obtain set-views of the hash table, which permits the use of iterators. To do so, you simply use one of the collection-view methods defined by Map, such as entrySet( ) or keySet( ). For example, you can obtain a set-view of the keys and cycle through them using either an iterator or an enhanced for loop. Here is a reworked version of the program that shows this technique:

```java
// Use iterators with a Hashtable.
import java.util.*;

class HTDemo2 {
    public static void main(String args[]) {
        Hashtable<String, Double> balance =
            new Hashtable<String, Double>();
        String str;
        double bal;

        balance.put("John Doe", 3434.34);
        balance.put("Tom Smith", 123.22);
        balance.put("Jane Baker", 1378.00);
        balance.put("Tod Hall", 99.22);
        balance.put("Ralph Smith", -19.08);
    }
}
```
Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. The Properties class is used by many other Java classes. For example, it is the type of object returned by `System.getProperties()` when obtaining environmental values. Although the Properties class, itself, is not generic, several of its methods are.

Properties defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a Properties object. Properties defines these constructors:

Properties( )
The first version creates a Properties object that has no default values. The second creates an object that uses propDefault for its default values. In both cases, the property list is empty.

In addition to the methods that Properties inherits from Hashtable, Properties defines the methods listed in Table 17-19. Properties also contains one deprecated method: save( ). This was replaced by store( ) because save( ) did not handle errors correctly.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getProperty(String key)</td>
<td>Returns the value associated with key. A null object is returned if key is neither in the list nor in the default property list.</td>
</tr>
<tr>
<td>String getProperty(String key, String defaultProperty)</td>
<td>Returns the value associated with key. defaultProperty is returned if key is neither in the list nor in the default property list.</td>
</tr>
<tr>
<td>void list(PrintStream streamOut)</td>
<td>Sends the property list to the output stream linked to streamOut.</td>
</tr>
<tr>
<td>void list(PrintWriter streamOut)</td>
<td>Sends the property list to the output stream linked to streamOut.</td>
</tr>
<tr>
<td>void load(InputStream streamIn) throws IOException</td>
<td>Inputs a property list from the input stream linked to streamIn.</td>
</tr>
<tr>
<td>void load(Reader streamIn) throws IOException</td>
<td>Inputs a property list from the input stream linked to streamIn.</td>
</tr>
<tr>
<td>void loadFromXML(InputStream streamIn) throws IOException, InvalidPropertiesFormatException</td>
<td>Inputs a property list from an XML document linked to streamIn.</td>
</tr>
</tbody>
</table>
One useful capability of the Properties class is that you can specify a default property that will be returned if no value is associated with a certain key. For example, a default value can be specified along with the key in the `getProperty()` method—such as `getProperty("name", "default value")`. If the “name” value is not found, then “default value” is returned. When you construct a Properties object, you can pass another instance of Properties to be used as the default properties for the new instance. In this case, if you call `getProperty("foo")` on a given Properties object, and “foo” does not exist, Java looks for “foo” in the default Properties object. This allows for arbitrary nesting of levels of default properties.

The following example demonstrates Properties. It creates a property list in which the keys are the names of
The output from this program is shown here:

The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.

Since Florida is not in the list, the default value is used.
Although it is perfectly valid to use a default value when you call `getProperty()`, as the preceding example shows, there is a better way of handling default values for most applications of property lists. For greater flexibility, specify a default property list when constructing a `Properties` object. The default list will be searched if the desired key is not found in the main list. For example, the following is a slightly reworked version of the preceding program, with a default list of states specified. Now, when Florida is sought, it will be found in the default list:

```java
// Use a default property list.
import java.util.*;

class PropDemoDef {
    public static void main(String args[]) {
        Properties defList = new Properties();
        defList.put("Florida", "Tallahassee");
        defList.put("Wisconsin", "Madison");

        Properties capitals = new Properties(defList);
        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        // Get a set-view of the keys.
        Set<?> states = capitals.keySet();

        // Show all of the states and capitals.
        for(Object name : states)
            System.out.println("The capital of " +
                                name + " is " +
                                capitals.getProperty((String)name) + ".");

        System.out.println();
    }
}
```
String str = capitals.getProperty("Florida");
System.out.println("The capital of Florida is " + str + ".");
}

Using store( ) and load( )

One of the most useful aspects of Properties is that the information contained in a Properties object can be easily stored to or loaded from disk with the store( ) and load( ) methods. At any time, you can write a Properties object to a stream or read it back. This makes property lists especially convenient for implementing simple databases. For example, the following program uses a property list to create a simple computerized telephone book that stores names and phone numbers. To find a person’s number, you enter his or her name. The program uses the store( ) and load( ) methods to store and retrieve the list. When the program executes, it first tries to load the list from a file called phonebook.dat. If this file exists, the list is loaded. You can then add to the list. If you do, the new list is saved when you terminate the program. Notice how little code is required to implement a small, but functional, computerized phone book.
// A simple telephone number database that uses
// a property list. */
import java.io.*;
import java.util.*;

class Phonebook {
    public static void main(String args[])
        throws IOException
    {
        Properties ht = new Properties();
        BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
        String name, number;
        FileInputStream fin = null;
        boolean changed = false;

        // Try to open phonebook.dat file.
        try {
            fin = new FileInputStream("phonebook.dat");
        } catch(FileNotFoundException e) {
            // ignore missing file
        }

        /* If phonebook file already exists,
         load existing telephone numbers. */
        try {
            if(fin != null) {
                ht.load(fin);
                fin.close();
            }
        } catch(IOException e) {
            System.out.println("Error reading file.");
        }

        // Let user enter new names and numbers.
        do {
            System.out.println("Enter new name" +
                    " ('quit' to stop): ");
            name = br.readLine();
            if(name.equals("quit")) continue;

            System.out.println("Enter number: ");
            number = br.readLine();
        } while

        /* Write new name and number to phonebook.dat file.
         */
        try {
            ht.setProperty(name, number);
            ht.store(fin, null);
        } catch(EOFException e) {
            System.out.println("EOF encountered.");
        }
    }
}
Parting Thoughts on Collections

The Collections Framework gives you, the programmer, a powerful set of well-engineered solutions to some of programming’s most common tasks. Consider using a collection the next time that you need to store and retrieve information. Remember, collections need not be reserved for only the “large jobs,” such as corporate databases, mailing lists, or inventory systems. They are also effective when applied to smaller jobs. For example, a TreeMap might make an excellent collection to hold the directory structure of a set of files. A TreeSet could be quite useful for storing project-management information. Frankly, the types of problems that will benefit from a collections-based solution are limited only by your imagination.
CHAPTER 18
java.util Part 2: More Utility Classes

This chapter continues our discussion of java.util by examining those classes and interfaces that are not part of the Collections Framework. These include classes that tokenize strings, work with dates, compute random numbers, bundle resources, and observe events. Also covered are the Formatter and Scanner classes which make it easy to write and read formatted data. Finally, the subpackages of java.util are briefly mentioned at the end of this chapter.

StringTokenizer
The processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts, or tokens, which in a certain sequence can convey a semantic meaning. The StringTokenizer class provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner. StringTokenizer implements the Enumeration interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using StringTokenizer.

To use StringTokenizer, you specify an input string and a string that contains delimiters. Delimiters are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, ",;:" sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage
The StringTokenizer constructors are shown here:

```java
StringTokenizer(String str)
StringTokenizer(String str, String delimiters)
StringTokenizer(String str, String delimiters, boolean delimAsToken)
```

In all versions, `str` is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, `delimiters` is a string that specifies the delimiters. In the third version, if `delimAsToken` is true, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

Once you have created a StringTokenizer object, the `nextToken()` method is used to extract consecutive tokens. The `hasMoreTokens()` method returns true while there are more tokens to be extracted. Since `StringTokenizer` implements `Enumeration`, the `hasMoreElements()` and `nextElement()` methods are also implemented, and they act the same as `hasMoreTokens()` and `nextToken()`, respectively. The StringTokenizer methods are shown in Table 18-1.

Here is an example that creates a StringTokenizer to parse “key=value” pairs. Consecutive sets of “key=value” pairs are separated by a semicolon.

```java
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;
```
class STDemo {
    static String in = "title=Java: The Complete Reference;" +
        "author=Schildt;" +
        "publisher=McGraw-Hill;" +
        "copyright=2011";

    public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer(in, ",;");

        while (st.hasMoreTokens()) {
            String key = st.nextToken();
            String val = st.nextToken();
            System.out.println(key + \t + val);
        }
    }
}

The output from this program is shown here:

<table>
<thead>
<tr>
<th>title</th>
<th>Java: The Complete Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>author</td>
<td>Schildt</td>
</tr>
<tr>
<td>publisher</td>
<td>McGraw-Hill</td>
</tr>
<tr>
<td>copyright</td>
<td>2011</td>
</tr>
</tbody>
</table>

Table 18-1 The Methods Defined by StringTokenizer

BitSet

A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed. This makes it similar to a vector of bits. The BitSet constructors are shown here:
BitSet( )
BitSet(int size)

The first version creates a default object. The second version allows you to specify its initial size (that is, the number of bits that it can hold). All bits are initialized to zero.

BitSet defines the methods listed in Table 18-2.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void and(BitSet bitSet)</td>
<td>ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object.</td>
</tr>
<tr>
<td>void andNot(BitSet bitSet)</td>
<td>For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.</td>
</tr>
<tr>
<td>int cardinality( )</td>
<td>Returns the number of set bits in the invoking object.</td>
</tr>
<tr>
<td>void clear( )</td>
<td>Zeros all bits.</td>
</tr>
<tr>
<td>void clear(int index)</td>
<td>Zeros the bit specified by index.</td>
</tr>
<tr>
<td>void clear(int startIndex, int endIndex)</td>
<td>Zeros the bits from startIndex to endIndex−1.</td>
</tr>
<tr>
<td>Object clone( )</td>
<td>Duplicates the invoking BitSet object.</td>
</tr>
<tr>
<td>boolean equals(Object bitSet)</td>
<td>Returns true if the invoking bit set is equivalent to the one passed in bitSet. Otherwise, the method returns false.</td>
</tr>
<tr>
<td>void flip(int index)</td>
<td>Reverses the bit specified by index.</td>
</tr>
<tr>
<td>void flip(int startIndex, int endIndex)</td>
<td>Reverses the bits from startIndex to endIndex−1.</td>
</tr>
<tr>
<td>boolean get(int index)</td>
<td>Returns the current state of the bit at the specified index.</td>
</tr>
<tr>
<td>BitSet get(int startIndex, int endIndex)</td>
<td>Returns a BitSet that consists of the bits from startIndex to endIndex−1. The invoking object is not changed.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Returns the hash code for the invoking object.</td>
</tr>
<tr>
<td>boolean intersects(BitSet bitSet)</td>
<td>Returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Returns true if all bits in the invoking object are zero.</td>
</tr>
<tr>
<td>int length()</td>
<td>Returns the number of bits required to hold the contents of the invoking BitSet. This value is determined by the location of the last 1 bit.</td>
</tr>
<tr>
<td>int nextClearBit(int startIndex)</td>
<td>Returns the index of the next cleared bit (that is, the next zero bit), starting from the index specified by startIndex.</td>
</tr>
<tr>
<td>int nextSetBit(int startIndex)</td>
<td>Returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, -1 is returned.</td>
</tr>
<tr>
<td>void or(BitSet bitSet)</td>
<td>ORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.</td>
</tr>
<tr>
<td>int previousClearBit(int startIndex)</td>
<td>Returns the index of the next cleared bit (that is, the next 0 bit) at or prior to the index specified by startIndex. If no cleared bit is found, -1 is returned. (Added by JDK 7.)</td>
</tr>
<tr>
<td>int previousSetBit(int startIndex)</td>
<td>Returns the index of the next set bit (that is, the next 1 bit) at or prior to the index specified by startIndex. If no set bit is found, -1 is returned. (Added by JDK 7.)</td>
</tr>
<tr>
<td>void set(int index)</td>
<td>Sets the bit specified by index.</td>
</tr>
<tr>
<td>void set(int index, boolean v)</td>
<td>Sets the bit specified by index to the value passed in v. true sets the bit; false clears the bit.</td>
</tr>
<tr>
<td>void set(int startIndex, int endIndex)</td>
<td>Sets the bits from startIndex to endIndex−1.</td>
</tr>
<tr>
<td>void set(int startIndex, int endIndex, boolean v)</td>
<td>Sets the bits from startIndex to endIndex−1 to the value passed in v. true sets the bits; false clears the bits.</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the number of bits in the invoking BitSet object.</td>
</tr>
<tr>
<td>byte[] toByteArray()</td>
<td>Returns a byte array that contains the invoking BitSet object. (Added by JDK 7.)</td>
</tr>
<tr>
<td>long[] toLongArray()</td>
<td>Returns a long array that contains the invoking BitSet object. (Added by JDK 7.)</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns the string equivalent of the invoking BitSet object.</td>
</tr>
<tr>
<td>static BitSet valueOf(byte[] v)</td>
<td>Returns a BitSet that contains the bits in v. (Added by JDK 7.)</td>
</tr>
<tr>
<td>static BitSet valueOf(ByteBuffer v)</td>
<td>Returns a BitSet that contains the bits in v. (Added by JDK 7.)</td>
</tr>
<tr>
<td>static BitSet valueOf(long[] v)</td>
<td>Returns a BitSet that contains the bits in v. (Added by JDK 7.)</td>
</tr>
<tr>
<td>static BitSet valueOf(LongBuffer v)</td>
<td>Returns a BitSet that contains the bits in v. (Added by JDK 7.)</td>
</tr>
<tr>
<td>void xor(BitSet bitSet)</td>
<td>XORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.</td>
</tr>
</tbody>
</table>
Here is an example that demonstrates BitSet:

```java
// BitSet Demonstration.
import java.util.BitSet;

class BitSetDemo {
    public static void main(String args[]) {
        BitSet bits1 = new BitSet(16);
        BitSet bits2 = new BitSet(16);

        // set some bits
        for(int i=0; i<16; i++) {
            if((i%2) == 0) bits1.set(i);
            if((i%5) != 0) bits2.set(i);
        }

        System.out.println("Initial pattern in bits1: ");
        System.out.println(bits1);
        System.out.println("Initial pattern in bits2: ");
        System.out.println(bits2);

        // AND bits
        bits2.and(bits1);
        System.out.println("\nbits2 AND bits1: ");
        System.out.println(bits2);

        // OR bits
        bits2.or(bits1);
        System.out.println("\nbits2 OR bits1: ");
        System.out.println(bits2);

        // XOR bits
        bits2.xor(bits1);
        System.out.println("\nbits2 XOR bits1: ");
        System.out.println(bits2);
    }
}
```

The output from this program is shown here. When `toString()` converts a BitSet object to its string equivalent, each set bit is represented by its bit position. Cleared bits are not shown.
Date
The Date class encapsulates the current date and time. Before beginning our examination of Date, it is important to point out that it has changed substantially from its original version defined by Java 1.0. When Java 1.1 was released, many of the functions carried out by the original Date class were moved into the Calendar and DateFormat classes, and as a result, many of the original 1.0 Date methods were deprecated. Since the deprecated 1.0 methods should not be used for new code, they are not described here.

Date supports the following constructors:

Date()
Date(long millisec)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970. The nondeprecated methods defined by Date are shown in Table 18-3. Date also implements the Comparable
Table 18-3 The Nondeprecated Methods Defined by Date

As you can see by examining Table 18-3, the non-deprecated Date features do not allow you to obtain the individual components of the date or time. As the following program demonstrates, you can only obtain the date and time in terms of milliseconds or in its default string representation as returned by \texttt{toString()}.

To obtain more-detailed information about the date and time, you will use the Calendar class.

```java
// Show date and time using only Date methods.
import java.util.Date;

class DateDemo {
    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date using toString()
        
```
System.out.println(date);

// Display number of milliseconds since midnight, January 1, 1970 GMT
long msec = date.getTime();
System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
}
}

Sample output is shown here:

Sat Jan 01 10:27:33 CST 2011
Milliseconds since Jan. 1, 1970 GMT = 1293899253417

Calendar
The abstract Calendar class provides a set of methods that allows you to convert a time in milliseconds to a number of useful components. Some examples of the type of information that can be provided are year, month, day, hour, minute, and second. It is intended that subclasses of Calendar will provide the specific functionality to interpret time information according to their own rules. This is one aspect of the Java class library that enables you to write programs that can operate in international environments. An example of such a subclass is GregorianCalendar.

Calendar provides no public constructors.

Calendar defines several protected instance variables. areFieldsSet is a boolean that indicates if the time components have been set. fields is an array of ints that holds the components of the time. isSet is a boolean array that indicates if a specific time component has been set. time is a long that holds the current time for this object. isTimeSet is a boolean that indicates if the current time has been set.
Some commonly used methods defined by Calendar are shown in **Table 18-4**.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abstract void add(int which, int val)</code></td>
<td>Adds <code>val</code> to the time or date component specified by <code>which</code>. To subtract, add a negative value. <code>which</code> must be one of the fields defined by <code>Calendar</code>, such as <code>Calendar.HOUR</code>.</td>
</tr>
<tr>
<td><code>boolean after(Object calendarObj)</code></td>
<td>Returns <code>true</code> if the invoking <code>Calendar</code> object contains a date that is later than the one specified by <code>calendarObj</code>. Otherwise, it returns <code>false</code>.</td>
</tr>
<tr>
<td><code>boolean before(Object calendarObj)</code></td>
<td>Returns <code>true</code> if the invoking <code>Calendar</code> object contains a date that is earlier than the one specified by <code>calendarObj</code>. Otherwise, it returns <code>false</code>.</td>
</tr>
<tr>
<td><code>final void clear()</code></td>
<td>Zeros all time components in the invoking object.</td>
</tr>
<tr>
<td><code>final void clear(int which)</code></td>
<td>Zeros the time component specified by <code>which</code> in the invoking object.</td>
</tr>
<tr>
<td><code>Object clone()</code></td>
<td>Returns a duplicate of the invoking object.</td>
</tr>
<tr>
<td><code>boolean equals(Object calendarObj)</code></td>
<td>Returns <code>true</code> if the invoking <code>Calendar</code> object contains a date that is equal to the one specified by <code>calendarObj</code>. Otherwise, it returns <code>false</code>.</td>
</tr>
<tr>
<td><code>int get(int calendarField)</code></td>
<td>Returns the value of one component of the invoking object. The component is indicated by <code>calendarField</code>. Examples of the components that can be requested are <code>Calendar.YEAR</code>, <code>Calendar.MONTH</code>, <code>Calendar.MINUTE</code>, and so forth.</td>
</tr>
<tr>
<td><code>static Locale[] getAvailableLocales()</code></td>
<td>Returns an array of <code>Locale</code> objects that contains the locales for which calendars are available.</td>
</tr>
<tr>
<td><code>static Calendar getInstance()</code></td>
<td>Returns a <code>Calendar</code> object for the default locale and time zone.</td>
</tr>
<tr>
<td><code>static Calendar getInstance(TimeZone tz)</code></td>
<td>Returns a <code>Calendar</code> object for the time zone specified by <code>tz</code>. The default locale is used.</td>
</tr>
<tr>
<td><code>static Calendar getInstance(Locale locale)</code></td>
<td>Returns a <code>Calendar</code> object for the locale specified by <code>locale</code>. The default time zone is used.</td>
</tr>
<tr>
<td><code>static Calendar getInstance(TimeZone tz, Locale locale)</code></td>
<td>Returns a <code>Calendar</code> object for the time zone specified by <code>tz</code> and the locale specified by <code>locale</code>.</td>
</tr>
<tr>
<td><code>final Date getTime()</code></td>
<td>Returns a <code>Date</code> object equivalent to the time of the invoking object.</td>
</tr>
<tr>
<td><code>TimeZone getTimeZone()</code></td>
<td>Returns the time zone for the invoking object.</td>
</tr>
<tr>
<td><code>final boolean isSet(int which)</code></td>
<td>Returns <code>true</code> if the specified time component is set. Otherwise, it returns <code>false</code>.</td>
</tr>
</tbody>
</table>
Table 18-4 Commonly Used Methods Defined by Calendar

Calendar defines the following int constants, which are used when you get or set components of the calendar:

<table>
<thead>
<tr>
<th>ALL_STYLES</th>
<th>FRIDAY</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>HOUR</td>
<td>SATURDAY</td>
</tr>
<tr>
<td>AM_PM</td>
<td>HOUR_OF_DAY</td>
<td>SECOND</td>
</tr>
<tr>
<td>APRIL</td>
<td>JANUARY</td>
<td>SEPTEMBER</td>
</tr>
<tr>
<td>AUGUST</td>
<td>JULY</td>
<td>SHORT</td>
</tr>
<tr>
<td>DATE</td>
<td>JUNE</td>
<td>SUNDAY</td>
</tr>
<tr>
<td>DAY_OF_MONTH</td>
<td>LONG</td>
<td>THURSDAY</td>
</tr>
<tr>
<td>DAY_OF_WEEK</td>
<td>MARCH</td>
<td>TUESDAY</td>
</tr>
<tr>
<td>DAY_OF_WEEK_IN_MONTH</td>
<td>MAY</td>
<td>UNDECEMBER</td>
</tr>
<tr>
<td>DAY_OF_YEAR</td>
<td>MILLISECOND</td>
<td>WEDNESDAY</td>
</tr>
<tr>
<td>DECEMBER</td>
<td>MINUTE</td>
<td>WEEK_OF_MONTH</td>
</tr>
<tr>
<td>DST_OFFSET</td>
<td>MONDAY</td>
<td>WEEK_OF_YEAR</td>
</tr>
<tr>
<td>ERA</td>
<td>MONTH</td>
<td>YEAR</td>
</tr>
<tr>
<td>FEBRUARY</td>
<td>NOVEMBER</td>
<td>ZONE_OFFSET</td>
</tr>
<tr>
<td>FIELD_COUNT</td>
<td>OCTOBER</td>
<td></td>
</tr>
</tbody>
</table>
// Demonstrate Calendar
import java.util.Calendar;

class CalendarDemo {
    public static void main(String args[]) {
        String months[] = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};

        // Create a calendar initialized with the current date and time in the default locale and timezone.
        Calendar calendar = Calendar.getInstance();

        // Display current time and date information.
        System.out.print("Date: ");
        System.out.print(months[calendar.get(Calendar.MONTH)]);
        System.out.print(" "+calendar.get(Calendar.DATE)+" ");
        System.out.println(calendar.get(Calendar.YEAR));

        System.out.print("Time: ");
        System.out.print(calendar.get(Calendar.HOUR)+":");
        System.out.print(calendar.get(Calendar.MINUTE)+":");
        System.out.println(calendar.get(Calendar.SECOND));

        // Set the time and date information and display it.
        calendar.set(Calendar.HOUR, 10);
        calendar.set(Calendar.MINUTE, 29);
        calendar.set(Calendar.SECOND, 22);
        System.out.print(\"Updated time: ");
        System.out.print(calendar.get(Calendar.HOUR)+":");
        System.out.print(calendar.get(Calendar.MINUTE)+":");
        System.out.println(calendar.get(Calendar.SECOND));
    }
}

Sample output is shown here:

Date: Jan 1 2011
Time: 11:24:25
Updated time: 10:29:22

GregorianCalendar
GregorianCalendar is a concrete implementation of a
Calendar that implements the normal Gregorian calendar with which you are familiar. The getInstance() method of Calendar will typically return a GregorianCalendar initialized with the current date and time in the default locale and time zone.

GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for GregorianCalendar objects. The default, GregorianCalendar(), initializes the object with the current date and time in the default locale and time zone. Three more constructors offer increasing levels of specificity:

```java
GregorianCalendar(int year, int month, int dayOfMonth)
GregorianCalendar(int year, int month, int dayOfMonth, int hours, int minutes)
GregorianCalendar(int year, int month, int dayOfMonth, int hours, int minutes, int seconds)
```

All three versions set the day, month, and year. Here, year specifies the year. The month is specified by month, with zero indicating January. The day of the month is specified by dayOfMonth. The first version sets the time to midnight. The second version also sets the hours and the minutes. The third version adds seconds.
You can also construct a GregorianCalendar object by specifying the locale and/or time zone. The following constructors create objects initialized with the current date and time using the specified time zone and/or locale:

- `GregorianCalendar(Locale locale)`
- `GregorianCalendar(TimeZone timeZone)`
- `GregorianCalendar(TimeZone timeZone, Locale locale)`

GregorianCalendar provides an implementation of all the abstract methods in Calendar. It also provides some additional methods. Perhaps the most interesting is `isLeapYear()`, which tests if the year is a leap year. Its form is

```java
boolean isLeapYear(int year)
```

This method returns true if year is a leap year and false otherwise.

The following program demonstrates GregorianCalendar:
import java.util.*;

class GregorianCalendarDemo {
    public static void main(String args[]) {
        String months[] = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};

        int year;

        // Create a Gregorian calendar initialized
        // with the current date and time in the
        // default locale and timezone.
        GregorianCalendar gcalendar = new GregorianCalendar();

        // Display current time and date information.
        System.out.print("Date: ");
        System.out.print(months[gcalendar.get(Calendar.MONTH)]);
        System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
        System.out.println(year = gcalendar.get(Calendar.YEAR));

        System.out.print("Time: ");
        System.out.print(gcalendar.get(Calendar.HOUR) + ":");
        System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
        System.out.print(gcalendar.get(Calendar.SECOND));

        // Test if the current year is a leap year
        if(gcalendar.isLeapYear(year)) {
            System.out.println("The current year is a leap year");
        } else {
            System.out.println("The current year is not a leap year");
        }
    }
}

Sample output is shown here:

Date: Jan 1 2011
Time: 11:25:27
The current year is not a leap year

TimeZone
Another time-related class is `TimeZone`. The abstract `TimeZone` class allows you to work with time zone offsets from Greenwich mean time (GMT), also referred to as Coordinated Universal Time (UTC). It also computes daylight saving time. `TimeZone` only supplies the default constructor.

A sampling of methods defined by `TimeZone` is given in Table 18-5.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clone()</code></td>
<td>Returns a <code>TimeZone</code>-specific version of <code>clone()</code>.</td>
</tr>
<tr>
<td><code>getAvailableIDs()</code></td>
<td>Returns an array of <code>String</code> objects representing the names of all time zones.</td>
</tr>
<tr>
<td><code>getAvailableIDs(int timeDelta)</code></td>
<td>Returns an array of <code>String</code> objects representing the names of all time zones that are <code>timeDelta</code> offset from GMT.</td>
</tr>
<tr>
<td><code>getDefault()</code></td>
<td>Returns a <code>TimeZone</code> object that represents the default time zone used on the host computer.</td>
</tr>
<tr>
<td><code>getID()</code></td>
<td>Returns the name of the invoking <code>TimeZone</code> object.</td>
</tr>
<tr>
<td><code>getOffset(int era, int year, int month, int dayOfMonth, int dayOfWeek, int millis)</code></td>
<td>Returns the offset that should be added to GMT to compute local time. This value is adjusted for daylight saving time. The parameters to the method represent date and time components.</td>
</tr>
<tr>
<td><code>getRawOffset()</code></td>
<td>Returns the raw offset (in milliseconds) that should be added to GMT to compute local time. This value is not adjusted for daylight saving time.</td>
</tr>
<tr>
<td><code>getTimeZone(String tzName)</code></td>
<td>Returns the <code>TimeZone</code> object for the time zone named <code>tzName</code>.</td>
</tr>
<tr>
<td><code>inDaylightTime(Date d)</code></td>
<td>Returns <code>true</code> if the date represented by <code>d</code> is in daylight saving time in the invoking object. Otherwise, it returns <code>false</code>.</td>
</tr>
<tr>
<td><code>setDefault(TimeZone tz)</code></td>
<td>Sets the default time zone to be used on this host. <code>tz</code> is a reference to the <code>TimeZone</code> object to be used.</td>
</tr>
<tr>
<td><code>setID(String tzName)</code></td>
<td>Sets the name of the time zone (that is, its ID) to that specified by <code>tzName</code>.</td>
</tr>
<tr>
<td><code>setRawOffset(int millis)</code></td>
<td>Sets the offset in milliseconds from GMT.</td>
</tr>
<tr>
<td><code>useDaylightTime()</code></td>
<td>Returns <code>true</code> if the invoking object uses daylight saving time. Otherwise, it returns <code>false</code>.</td>
</tr>
</tbody>
</table>

Table 18-5 Some of the Methods Defined by `TimeZone`
The SimpleTimeZone class is a convenient subclass of TimeZone. It implements TimeZone’s abstract methods and allows you to work with time zones for a Gregorian calendar. It also computes daylight saving time.

SimpleTimeZone defines four constructors. One is

SimpleTimeZone(int timeDelta, String tzName)

This constructor creates a SimpleTimeZone object. The offset relative to Greenwich mean time (GMT) is timeDelta. The time zone is named tzName.

The second SimpleTimeZone constructor is

SimpleTimeZone(int timeDelta, String tzId, int dstMonth0, int dstDayInMonth0, int dstDay0, int time0, int dstMonth1, int dstDayInMonth1, int dstDay1, int time1)

Here, the offset relative to GMT is specified in timeDelta. The time zone name is passed in tzId. The start of daylight saving time is indicated by the parameters dstMonth0, dstDayInMonth0, dstDay0, and time0. The end of daylight saving time is indicated by the parameters dstMonth1, dstDayInMonth1, dstDay1, and time1.

The third SimpleTimeZone constructor is

SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
Here, dstDelta is the number of milliseconds saved during daylight saving time.

The fourth SimpleTimeZone constructor is

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
                int dstDayInMonth0, int dstDay0, int time0,
                int time0mode, int dstMonth1, int dstDayInMonth1,
                int dstDay1, int time1, int time1mode, int dstDelta)
```

Here, time0mode specifies the mode of the starting time, and time1mode specifies the mode of the ending time. Valid mode values include

<table>
<thead>
<tr>
<th>STANDARD_TIME</th>
<th>WALL_TIME</th>
<th>UTC_TIME</th>
</tr>
</thead>
</table>

The time mode indicates how the time values are interpreted. The default mode used by the other constructors is WALL_TIME.

Locale

The Locale class is instantiated to produce objects that describe a geographical or cultural region. It is one of
several classes that provide you with the ability to write programs that can execute in different international environments. For example, the formats used to display dates, times, and numbers are different in various regions.

Internationalization is a large topic that is beyond the scope of this book. However, many programs will only need to deal with its basics, which include setting the current locale.

The `Locale` class defines the following constants that are useful for dealing with the most common locales:

<table>
<thead>
<tr>
<th>CANADA</th>
<th>GERMAN</th>
<th>KOREAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANADA_FRENCH</td>
<td>GERMANY</td>
<td>PRC</td>
</tr>
<tr>
<td>CHINA</td>
<td>ITALIAN</td>
<td>SIMPLIFIED_CHINESE</td>
</tr>
<tr>
<td>CHINESE</td>
<td>ITALY</td>
<td>TAIWAN</td>
</tr>
<tr>
<td>ENGLISH</td>
<td>JAPAN</td>
<td>TRADITIONAL_CHINESE</td>
</tr>
<tr>
<td>FRANCE</td>
<td>JAPANESE</td>
<td>UK</td>
</tr>
<tr>
<td>FRENCH</td>
<td>KOREA</td>
<td>US</td>
</tr>
</tbody>
</table>

For example, the expression `Locale.CANADA` represents the `Locale` object for Canada.

The constructors for `Locale` are

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

These constructors build a `Locale` object to represent a specific language and in the case of the last two, country. These values must contain standard language and country codes. Auxiliary variant information can be provided in `variant`. 
Locale defines several methods. One of the most important is `setDefault()`, shown here:

```java
static void setDefault(Locale localeObj)
```

This sets the default locale used by the JVM to that specified by `localeObj`.

Some other interesting methods are the following:

```java
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

These return human-readable strings that can be used to display the name of the country, the name of the language, and the complete description of the locale.

The default locale can be obtained using `getDefault()`, shown here:

```java
static Locale getDefault()
```

JDK 7 adds significant upgrades to the `Locale` class that support Internet Engineering Task Force (IETF) BCP 47, which defines tags for identifying languages, and Unicode Technical Standard (UTS) 35, which defines the `Locale Data Markup Language (LDML)`. Support for BCP 47 and UTS 35 caused several features to be added to `Locale`, including several new methods and the `Locale.Builder` class. Among others, new methods include `getScript()`, which obtains the locale’s script, and `toLanguageTag()`, which obtains a string that contains the locale’s language tag. The `Locale.Builder` class
constructs Locale instances. It ensures that a locale specification is well-formed as defined by BCP 47. (The Locale constructors do not provide such a check.)

Calendar and GregorianCalendar are examples of classes that operate in a locale-sensitive manner. DateFormat and SimpleDateFormat also depend on the locale.

Random

The Random class is a generator of pseudorandom numbers. These are called pseudorandom numbers because they are simply uniformly distributed sequences. Random defines the following constructors:

```
Random()
Random(long seed)
```

The first version creates a number generator that uses a reasonably unique seed. The second form allows you to specify a seed value manually.

If you initialize a Random object with a seed, you define the starting point for the random sequence. If you use the same seed to initialize another Random object, you will extract the same random sequence. If you want to generate different sequences, specify different seed values. One way to do this is to use the current time to seed a Random object. This approach reduces the possibility of getting repeated sequences.

The public methods defined by Random are shown in Table 18-6.

As you can see, there are seven types of random
numbers that you can extract from a Random object. Random Boolean values are available from nextBoolean(). Random bytes can be obtained by calling nextBytes(). Integers can be extracted via the nextInt() method. Long integers, uniformly distributed over their range, can be obtained with nextLong().

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean nextBoolean( )</td>
<td>Returns the next boolean random number.</td>
</tr>
<tr>
<td>void nextBytes(byte vals[])</td>
<td>Fills vals with randomly generated values.</td>
</tr>
<tr>
<td>double nextDouble( )</td>
<td>Returns the next double random number.</td>
</tr>
<tr>
<td>float nextFloat( )</td>
<td>Returns the next float random number.</td>
</tr>
<tr>
<td>double nextGaussian( )</td>
<td>Returns the next Gaussian random number.</td>
</tr>
<tr>
<td>int nextInt( )</td>
<td>Returns the next int random number.</td>
</tr>
<tr>
<td>int nextInt(int n)</td>
<td>Returns the next int random number within the range zero to n.</td>
</tr>
<tr>
<td>long nextLong( )</td>
<td>Returns the next long random number.</td>
</tr>
<tr>
<td>void setSeed(long newSeed)</td>
<td>Sets the seed value (that is, the starting point for the random number generator) to that specified by newSeed.</td>
</tr>
</tbody>
</table>

Table 18-6 The Methods Defined by Random

The nextFloat( ) and nextDouble( ) methods return a uniformly distributed float and double, respectively, between 0.0 and 1.0. Finally, nextGaussian( ) returns a double value centered at 0.0 with a standard deviation of 1.0. This is what is known as a bell curve.

Here is an example that demonstrates the sequence produced by nextGaussian( ). It obtains 100 random Gaussian values and averages these values. The program also counts the number of values that fall within two standard deviations, plus or minus, using increments of 0.5 for each category. The result is graphically displayed sideways on the screen.

```java
// Demonstrate random Gaussian values.
import java.util.Random;
```
Here is a sample program run. As you can see, a bell-like
distribution of numbers is obtained.

Average of values: 0.0702235271133344
**
*******
******
****************
*****************
*************
**********
********
***

Observable
The Observable class is used to create subclasses that
other parts of your program can observe. When an object of such a subclass undergoes a change, observing classes are notified. Observing classes must implement the Observer interface, which defines the update( ) method. The update( ) method is called when an observer is notified of a change in an observed object.

Observable defines the methods shown in Table 18-7. An object that is being observed must follow two simple rules. First, if it has changed, it must call setChanged( ). Second, when it is ready to notify observers of this change, it must call notifyObservers( ). This causes the update( ) method in the observing object(s) to be called. Be careful—if the object calls notifyObservers( ) without having previously called setChanged( ), no action will take place. The observed object must call both setChanged( ) and notifyObservers( ) before update( ) will be called.

Notice that notifyObservers( ) has two forms: one that takes an argument and one that does not. If you call notifyObservers( ) with an argument, this object is passed to the observer’s update( ) method as its second parameter. Otherwise, null is passed to update( ). You can use the second parameter for passing any type of object that is appropriate for your application.

The Observer Interface
To observe an observable object, you must implement the Observer interface. This interface defines only the one method shown here:
void update(Observable observOb, Object arg)

Here, observOb is the object being observed, and arg is the value passed by notifyObservers(). The update() method is called when a change in the observed object takes place.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addObserver(Observer obj)</td>
<td>Adds obj to the list of objects observing the invoking object.</td>
</tr>
<tr>
<td>protected void clearChanged()</td>
<td>Calling this method returns the status of the invoking object to &quot;unchanged.&quot;</td>
</tr>
<tr>
<td>int countObservers()</td>
<td>Returns the number of objects observing the invoking object.</td>
</tr>
<tr>
<td>void deleteObserver(Observer obj)</td>
<td>Removes obj from the list of objects observing the invoking object.</td>
</tr>
<tr>
<td>void deleteObservers()</td>
<td>Removes all observers for the invoking object.</td>
</tr>
<tr>
<td>boolean hasChanged()</td>
<td>Returns true if the invoking object has been modified and false if it has not.</td>
</tr>
<tr>
<td>void notifyObservers()</td>
<td>Notifies all observers of the invoking object that it has changed by calling update(). A null is passed as the second argument to update().</td>
</tr>
<tr>
<td>void notifyObservers(Object obj)</td>
<td>Notifies all observers of the invoking object that it has changed by calling update(). obj is passed as an argument to update().</td>
</tr>
<tr>
<td>protected void setChanged()</td>
<td>Called when the invoking object has changed.</td>
</tr>
</tbody>
</table>

Table 18-7 The Methods Defined by Observable

An Observer Example

Here is an example that demonstrates an observable object. It creates an observer class, called Watcher, that implements the Observer interface. The class being monitored is called BeingWatched. It extends Observable. Inside BeingWatched is the method counter(), which simply counts down from a specified value. It uses sleep() to wait a tenth of a second between counts. Each time the count changes, notifyObservers() is called with the current count passed as its argument. This causes the update() method inside Watcher to be called, which
displays the current count. Inside main( ), a Watcher and a BeingWatched object, called observing and observed, respectively, are created. Then, observing is added to the list of observers for observed. This means that observing.update( ) will be called each time counter( ) calls notifyObservers( ).

```java
/* Demonstrate the Observable class and the Observer interface. */
import java.util.*;

// This is the observing class.
class Watcher implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println("update() called, count is " + ((Integer)arg).intValue());
    }
}

// This is the class being observed.
class BeingWatched extends Observable {
    void counter(int period) {
        for(; period >= 0; period--) {
            setChanged();
            notifyObservers(new Integer(period));
            try {
                Thread.sleep(100);
            } catch(InterruptedException e) {
                System.out.println("Sleep interrupted");
            }
        }
    }
}
```
class ObserverDemo {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher observing = new Watcher();

        /* Add the observing to the list of observers for
          observed object. */
        observed.addObserver(observing);

        observed.counter(10);
    }
}

The output from this program is shown here:

    update() called, count is 10
    update() called, count is 9
    update() called, count is 8
    update() called, count is 7
    update() called, count is 6
    update() called, count is 5
    update() called, count is 4
    update() called, count is 3
    update() called, count is 2
    update() called, count is 1
    update() called, count is 0

More than one object can be an observer. For example, the following program implements two observing classes and adds an object of each class to the BeingWatched observer list. The second observer waits until the count reaches zero and then rings the bell.
/* An object may be observed by two or more observers. */

import java.util.*;

// This is the first observing class.
class Watcher1 implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println("update() called, count is " +
            ((Integer)arg).intValue());
    }
}

// This is the second observing class.
class Watcher2 implements Observer {
    public void update(Observable obj, Object arg) {
        // Ring bell when done
        if(((Integer)arg).intValue() == 0)
            System.out.println("Done" + '\7');
    }
}
The Observable class and the Observer interface allow you to implement sophisticated program architectures based on the document/view methodology. They are also useful in multithreaded situations.

Timer and TimerTask
An interesting and useful feature offered by java.util is the ability to schedule a task for execution at some future time. The classes that support this are Timer and TimerTask. Using these classes, you can create a thread that runs in the background, waiting for a specific time.
When the time arrives, the task linked to that thread is executed. Various options allow you to schedule a task for repeated execution, and to schedule a task to run on a specific date. Although it was always possible to manually create a task that would be executed at a specific time using the Thread class, Timer and TimerTask greatly simplify this process.

Timer and TimerTask work together. Timer is the class that you will use to schedule a task for execution. The task being scheduled must be an instance of TimerTask. Thus, to schedule a task, you will first create a TimerTask object and then schedule it for execution using an instance of Timer.

TimerTask implements the Runnable interface; thus, it can be used to create a thread of execution. Its constructor is shown here:

```java
TimerTask()
```

TimerTask defines the methods shown in Table 18-8. Notice that `run()` is abstract, which means that it must be overridden. The `run()` method, defined by the Runnable interface, contains the code that will be executed. Thus, the easiest way to create a timer task is to extend TimerTask and override `run()`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean cancel( )</td>
<td>Terminates the task. Returns true if an execution of the task is prevented. Otherwise, returns false.</td>
</tr>
<tr>
<td>abstract void run( )</td>
<td>Contains the code for the timer task.</td>
</tr>
<tr>
<td>long scheduledExecutionTime( )</td>
<td>Returns the time at which the last execution of the task was scheduled to have occurred.</td>
</tr>
</tbody>
</table>

Table 18-8 The Methods Defined by TimerTask
Once a task has been created, it is scheduled for execution by an object of type Timer. The constructors for Timer are shown here:

```
Timer( )
Timer(boolean DThread)
Timer(String tName)
Timer(String tName, boolean DThread)
```

The first version creates a Timer object that runs as a normal thread. The second uses a daemon thread if DThread is true. A daemon thread will execute only as long as the rest of the program continues to execute. The third and fourth constructors allow you to specify a name for the Timer thread. The methods defined by Timer are shown in Table 18-9.

Once a Timer has been created, you will schedule a task by calling schedule( ) on the Timer that you created. As Table 18-9 shows, there are several forms of schedule( ) which allow you to schedule tasks in a variety of ways.
If you create a non-daemon task, then you will want to call `cancel()` to end the task when your program ends. If you don’t do this, then your program may “hang” for a period of time.

The following program demonstrates `Timer` and `TimerTask`. It defines a timer task whose `run()` method displays the message “Timer task executed.” This task is scheduled to run once every half second after an initial delay of one second.

```java
// Demonstrate Timer and TimerTask.
```
class MyTimerTask extends TimerTask {
    public void run() {
        System.out.println("Timer task executed.");
    }
}

class TTest {
    public static void main(String args[]) {
        MyTimerTask myTask = new MyTimerTask();
        Timer myTimer = new Timer();

        /* Set an initial delay of 1 second, 
         * then repeat every half second. 
         */
        myTimer.schedule(myTask, 1000, 500);

        try {
            Thread.sleep(5000);
        } catch (InterruptedException exc) {}

        myTimer.cancel();
    }
}

Currency

The Currency class encapsulates information about a 
currency. It defines no constructors. The methods 
supported by Currency are shown in Table 18-10. The 
following program demonstrates Currency:

// Demonstrate Currency.
import java.util.*;

class CurDemo {
    public static void main(String args[]) {
        Currency c;

        c = Currency.getInstance(Locale.US);

        System.out.println("Symbol: " + c.getSymbol());
        System.out.println("Default fractional digits: " +
                        c.getDefaultFractionDigits());
    }
}
At the core of Java’s support for creating formatted output is the Formatter class. It provides format conversions that let you display numbers, strings, and time and date in virtually any format you like. It operates in a manner similar to the C/C++ printf() function, which means that if you are familiar with C/C++, then learning to use Formatter will be very
easy. It also further streamlines the conversion of C/C++ code to Java. If you are not familiar with C/C++, it is still quite easy to format data.

NOTE
Although Java’s `Formatter` class operates in a manner very similar to the C/C++ `printf()` function, there are some differences, and some new features. Therefore, if you have a C/C++ background, a careful reading is advised.

The Formatter Constructors
Before you can use `Formatter` to format output, you must create a `Formatter` object. In general, `Formatter` works by converting the binary form of data used by a program into formatted text. It stores the formatted text in a buffer, the contents of which can be obtained by your program whenever they are needed. It is possible to let `Formatter` supply this buffer automatically, or you can specify the buffer explicitly when a `Formatter` object is created. It is also possible to have `Formatter` output its buffer to a file.

The `Formatter` class defines many constructors, which enable you to construct a `Formatter` in a variety of ways. Here is a sampling:

```
Formatter()
Formatter(Appendable buf)
Formatter(Appendable buf, Locale loc)
```
Formatter(String filename)

    throws FileNotFoundException

Formatter(String filename, String charset)

    throws FileNotFoundException, UnsupportedEncodingException

Formatter(File outF)

    throws FileNotFoundException

Formatter(OutputStream outStrm)

Here, buf specifies a buffer for the formatted output. If buf is null, then Formatter automatically allocates a StringBuilder to hold the formatted output. The loc parameter specifies a locale. If no locale is specified, the default locale is used. The filename parameter specifies the name of a file that will receive the formatted output. The charset parameter specifies the character set. If no character set is specified, then the default character set is used. The outF parameter specifies a reference to an open file that will receive output. The outStrm parameter specifies a reference to an output stream that will receive output. When using a file, output is also written to the file.

Perhaps the most widely used constructor is the first, which has no parameters. It automatically uses the default locale and allocates a StringBuilder to hold the formatted output.
The Formatter Methods

Formatter defines the methods shown in Table 18-11.

Formatting Basics

After you have created a Formatter, you can use it to create a formatted string. To do so, use the format( ) method. The most commonly used version is shown here:

```java
Formatter format(String fmtString, Object ... args)
```

The fmtString consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains format specifiers that define the way the subsequent arguments are displayed.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close( )</td>
<td>Closes the invoking Formatter. This causes any resources used by the object to be released. After a Formatter has been closed, it cannot be reused. An attempt to use a closed Formatter results in a FormatterClosedException.</td>
</tr>
<tr>
<td>void flush( )</td>
<td>Flushes the format buffer. This causes any output currently in the buffer to be written to the destination. This applies mostly to a Formatter tied to a file.</td>
</tr>
<tr>
<td>Formatter format(String fmtString,</td>
<td>Formats the arguments passed via args according to the format specifiers contained in fmtString. Returns the invoking object.</td>
</tr>
<tr>
<td>Object ... args)</td>
<td></td>
</tr>
<tr>
<td>Formatter format(Locale loc,</td>
<td>Formats the arguments passed via args according to the format specifiers contained in fmtString. The locale specified by loc is used for this format. Returns the invoking object.</td>
</tr>
<tr>
<td>String fmtString, Object ... args)</td>
<td></td>
</tr>
<tr>
<td>IOException ioException( )</td>
<td>If the underlying object that is the destination for output throws an IOException, then this exception is returned. Otherwise, null is returned.</td>
</tr>
<tr>
<td>Locale locale( )</td>
<td>Returns the invoking object’s locale.</td>
</tr>
<tr>
<td>Appendable out( )</td>
<td>Returns a reference to the underlying object that is the destination for output.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns a String containing the formatted output.</td>
</tr>
</tbody>
</table>

Table 18-11 The Methods Defined by Formatter
In its simplest form, a format specifier begins with a percent sign followed by the format conversion specifier. All format conversion specifiers consist of a single character. For example, the format specifier for floating-point data is %f. In general, there must be the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order from left to right. For example, consider this fragment:

```java
Formatter fmt = new Formatter();
fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98);
```

This sequence creates a Formatter that contains the following string:

```plaintext
Formatting with Java is easy 10 98.600000
```

In this example, the format specifiers, %s, %d, and %f, are replaced with the arguments that follow the format string. Thus, %s is replaced by “with Java”, %d is replaced by 10, and %f is replaced by 98.6. All other characters are simply used as-is. As you might guess, the format specifier %s specifies a string, and %d specifies an integer value. As mentioned earlier, the %f specifies a floating-point value.

The format( ) method accepts a wide variety of format specifiers, which are shown in Table 18-12. Notice that many specifiers have both upper- and lowercase forms. When an uppercase specifier is used, then letters are shown in uppercase. Otherwise, the upper- and lowercase specifiers perform the same conversion. It is important to understand that Java type-checks each
format specifier against its corresponding argument. If the argument doesn’t match, an IllegalFormatException is thrown.

Once you have formatted a string, you can obtain it by calling toString(). For example, continuing with the preceding example, the following statement obtains the formatted string contained in fmt:

```java
String str = fmt.toString();
```

Of course, if you simply want to display the formatted string, there is no reason to first assign it to a String object. When a Formatter object is passed to println(), for example, its toString() method is automatically called.

Here is a short program that puts together all of the pieces, showing how to create and display a formatted string:

```java
// A very simple example that uses Formatter.
import java.util.*;

class FormatDemo {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.

        System.out.println(fmt);
        fmt.close();
    }
}
```
Table 18-12 The Format Specifiers

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Conversion Applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Floating-point hexadecimal</td>
</tr>
<tr>
<td>%A</td>
<td></td>
</tr>
<tr>
<td>%b</td>
<td>Boolean</td>
</tr>
<tr>
<td>%B</td>
<td></td>
</tr>
<tr>
<td>%c</td>
<td>Character</td>
</tr>
<tr>
<td>%d</td>
<td>Decimal integer</td>
</tr>
<tr>
<td>%e</td>
<td>Scientific notation</td>
</tr>
<tr>
<td>%E</td>
<td></td>
</tr>
<tr>
<td>%f</td>
<td>Decimal floating-point</td>
</tr>
<tr>
<td>%g</td>
<td>Uses %e or %f, whichever is shorter</td>
</tr>
<tr>
<td>%G</td>
<td></td>
</tr>
<tr>
<td>%o</td>
<td>Octal integer</td>
</tr>
<tr>
<td>%n</td>
<td>Inserts a newline character</td>
</tr>
<tr>
<td>%s</td>
<td>String</td>
</tr>
<tr>
<td>%S</td>
<td></td>
</tr>
<tr>
<td>%t</td>
<td>Time and date</td>
</tr>
<tr>
<td>%T</td>
<td></td>
</tr>
<tr>
<td>%x</td>
<td>Integer hexadecimal</td>
</tr>
<tr>
<td>%X</td>
<td></td>
</tr>
<tr>
<td>%%</td>
<td>Inserts a % sign</td>
</tr>
</tbody>
</table>

One other point: You can obtain a reference to the underlying output buffer by calling out(). It returns a reference to an Appendable object.

Now that you know the general mechanism used to create a formatted string, the remainder of this section discusses in detail each conversion. It also describes various options, such as justification, minimum field width, and precision.

Formatting Strings and Characters

To format an individual character, use %c. This causes the matching character argument to be output,
unmodified. To format a string, use %s.

Formatting Numbers

To format an integer in decimal format, use %d. To format a floating-point value in decimal format, use %f. To format a floating-point value in scientific notation, use %e. Numbers represented in scientific notation take this general form:

\[
 x.ddddddde +/- yy
\]

The %g format specifier causes Formatter to use either %f or %e, whichever is shorter. The following program demonstrates the effect of the %g format specifier:

```java
// Demonstrate the %g format specifier.
import java.util.*;

class FormatDemo2 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        for(double i=1000; i < 1.0e+10; i *= 100) {
            fmt.format("%g ", i);
            System.out.println(fmt);
        }
        fmt.close();
    }
}
```

It produces the following output:

```
1000.000000
1000.000000 100000.000000
1000.000000 100000.000000 1.000000e+07
1000.000000 100000.000000 1.000000e+07 1.000000e+09
```

You can display integers in octal or hexadecimal format by using %o and %x, respectively. For example,
You can display floating-point values in hexadecimal format by using %a. The format produced by %a appears a bit strange at first glance. This is because its representation uses a form similar to scientific notation that consists of a significand and an exponent, both in hexadecimal. Here is the general format:

0x1.sigpexp

Here, sig contains the fractional portion of the significand and exp contains the exponent. The p indicates the start of the exponent. For example, this call:

fmt.format("%a", 123.123);

produces this output:

0x1.ec7df3b645a1dp6

Formatting Time and Date

One of the more powerful conversion specifiers is %t. It lets you format time and date information. The %t specifier works a bit differently than the others because it requires the use of a suffix to describe the portion and precise format of the time or date desired. The suffixes are shown in **Table 18-13**. For example, to display minutes, you would use %tM, where M indicates minutes in a two-character field. The argument corresponding to
the %t specifier must be of type Calendar, Date, Long, or long.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Replaced By</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Abbreviated weekday name</td>
</tr>
<tr>
<td>A</td>
<td>Full weekday name</td>
</tr>
<tr>
<td>b</td>
<td>Abbreviated month name</td>
</tr>
<tr>
<td>B</td>
<td>Full month name</td>
</tr>
<tr>
<td>c</td>
<td>Standard date and time string formatted as day month date hh:mm:ss tz zone year</td>
</tr>
<tr>
<td>C</td>
<td>First two digits of year</td>
</tr>
<tr>
<td>d</td>
<td>Day of month as a decimal (01—31)</td>
</tr>
<tr>
<td>D</td>
<td>month/day/year</td>
</tr>
<tr>
<td>e</td>
<td>Day of month as a decimal (1—31)</td>
</tr>
<tr>
<td>F</td>
<td>year-month-day</td>
</tr>
<tr>
<td>h</td>
<td>Abbreviated month name</td>
</tr>
<tr>
<td>H</td>
<td>Hour (00 to 23)</td>
</tr>
<tr>
<td>I</td>
<td>Hour (01 to 12)</td>
</tr>
<tr>
<td>j</td>
<td>Day of year as a decimal (001 to 366)</td>
</tr>
<tr>
<td>k</td>
<td>Hour (0 to 23)</td>
</tr>
<tr>
<td>l</td>
<td>Hour (1 to 12)</td>
</tr>
<tr>
<td>L</td>
<td>Millisecond (000 to 999)</td>
</tr>
<tr>
<td>m</td>
<td>Month as decimal (01 to 13)</td>
</tr>
<tr>
<td>M</td>
<td>Minute as decimal (00 to 59)</td>
</tr>
<tr>
<td>N</td>
<td>Nanosecond (000000000 to 9999999999)</td>
</tr>
<tr>
<td>p</td>
<td>Locale’s equivalent of AM or PM in lowercase</td>
</tr>
<tr>
<td>Q</td>
<td>Milliseconds from 1/1/1970</td>
</tr>
<tr>
<td>r</td>
<td>hh:mm:ss (12-hour format)</td>
</tr>
<tr>
<td>R</td>
<td>hh:mm (24-hour format)</td>
</tr>
<tr>
<td>S</td>
<td>Seconds (00 to 60)</td>
</tr>
<tr>
<td>s</td>
<td>Seconds from 1/1/1970 UTC</td>
</tr>
<tr>
<td>T</td>
<td>hh:mm:ss (24-hour format)</td>
</tr>
<tr>
<td>y</td>
<td>Year in decimal without century (00 to 99)</td>
</tr>
<tr>
<td>Y</td>
<td>Year in decimal including century (0001 to 9999)</td>
</tr>
<tr>
<td>z</td>
<td>Offset from UTC</td>
</tr>
<tr>
<td>Z</td>
<td>Time zone name</td>
</tr>
</tbody>
</table>

Table 18-13 The Time and Date Format Suffixes
Here is a program that demonstrates several of the formats:

```java
// Formatting time and date.
import java.util.*;

class TimeDateFormat {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();

        // Display standard 12-hour time format.
        fmt.format("%tr", cal);
        System.out.println(fmt);
        fmt.close();

        // Display complete time and date information.
        fmt = new Formatter();
        fmt.format("%tc", cal);
        System.out.println(fmt);
        fmt.close();

        // Display just hour and minute.
        fmt = new Formatter();
        fmt.format("%tl:%tM", cal, cal);
        System.out.println(fmt);
        fmt.close();

        // Display month by name and number.
        fmt = new Formatter();
        fmt.format("%tB %tb %tm", cal, cal, cal);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Sample output is shown here:

09:17:15 AM
Sat Jan 01 09:17:15 CST 2011
9:17
January Jan 01

The %n and %%% Specifiers

The %n and %%% format specifiers differ from the others
in that they do not match an argument. Instead, they are simply escape sequences that insert a character into the output sequence. The `%n` inserts a newline. The `%%` inserts a percent sign. Neither of these characters can be entered directly into the format string. Of course, you can also use the standard escape sequence `\n` to embed a newline character.

Here is an example that demonstrates the `%n` and `%%` format specifiers:

```java
// Demonstrate the %n and %% format specifiers.
import java.util.*;
class FormatDemo3 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format("Copying file%nTransfer is %d%% complete", 88);
        System.out.println(fmt);
        fmt.close();
    }
}
```

It displays the following output:

```
Copying file
Transfer is 88% complete
```

### Specifying a Minimum Field Width

An integer placed between the `%` sign and the format conversion code acts as a minimum field-width specifier. This pads the output with spaces to ensure that it reaches a certain minimum length. If the string or number is longer than that minimum, it will still be printed in full. The default padding is done with spaces. If you want to pad with 0’s, place a 0 before the field-width specifier. For example, `%05d` will pad a number of less than five
digits with 0’s so that its total length is five. The field-width specifier can be used with all format specifiers except \%n.

The following program demonstrates the minimum field-width specifier by applying it to the %f conversion:

```java
// Demonstrate a field-width specifier.
import java.util.*;

class FormatDemo4 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format("|%f|%n|%12f|%n|%012f|",
                   10.12345, 10.12345, 10.12345);

        System.out.println(fmt);
        fmt.close();
    }
}
```

This program produces the following output:

```
|10.123450|
| 10.123450|
|00010.123450|
```

The first line displays the number 10.12345 in its default width. The second line displays that value in a 12-character field. The third line displays the value in a 12-character field, padded with leading zeros.

The minimum field-width modifier is often used to produce tables in which the columns line up. For example, the next program produces a table of squares and cubes for the numbers between 1 and 10:

```java
// Create a table of squares and cubes.
import java.util.*;
```
```java
class FieldWidthDemo {
    public static void main(String args[]) {
        Formatter fmt;

        for(int i=1; i <= 10; i++) {
            fmt = new Formatter();
            fmt.format("%4d %4d %4d", i, i*i, i*i*i);
            System.out.println(fmt);
            fmt.close();
        }
    }
}
```

Its output is shown here:

```
1     1    1
2     4    8
3     9   27
4    16   64
5    25  125
6    36  216
7    49  343
8    64  512
9    81  729
10   100 1000
```

Specifying Precision

A precision specifier can be applied to the %f, %e, %g, and %s format specifiers. It follows the minimum field-width specifier (if there is one) and consists of a period followed by an integer. Its exact meaning depends upon the type of data to which it is applied.

When you apply the precision specifier to floating-point data using the %f or %e specifiers, it determines the number of decimal places displayed. For example, %10.4f displays a number at least ten characters wide with four decimal places. When using %g, the precision determines the number of significant digits. The default precision is 6.
Applied to strings, the precision specifier specifies the maximum field length. For example, \%5.7s displays a string of at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated.

The following program illustrates the precision specifier:

```java
// Demonstrate the precision modifier.
import java.util.*;
class PrecisionDemo {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        // Format 4 decimal places.
        fmt.format("%.4f", 123.1234567);
        System.out.println(fmt);
        fmt.close();

        // Format to 2 decimal places in a 16 character field
        fmt = new Formatter();
        fmt.format("%16.2e", 123.1234567);
        System.out.println(fmt);
        fmt.close();

        // Display at most 15 characters in a string.
        fmt = new Formatter();
        fmt.format("%.15s", "Formatting with Java is now easy.");
        System.out.println(fmt);
        fmt.close();
    }
}
```

It produces the following output:

```
123.1235
  1.23e+02
Formatting with
```

Using the Format Flags

Formatter recognizes a set of format flags that lets you control various aspects of a conversion. All format flags
are single characters, and a format flag follows the % in a format specification. The flags are shown here:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>Left justification</td>
</tr>
<tr>
<td>#</td>
<td>Alternate conversion format</td>
</tr>
<tr>
<td>0</td>
<td>Output is padded with zeros rather than spaces</td>
</tr>
<tr>
<td>space</td>
<td>Positive numeric output is preceded by a space</td>
</tr>
<tr>
<td>+</td>
<td>Positive numeric output is preceded by a + sign</td>
</tr>
<tr>
<td>,</td>
<td>Numeric values include grouping separators</td>
</tr>
<tr>
<td>(</td>
<td>Negative numeric values are enclosed within parentheses</td>
</tr>
</tbody>
</table>

Not all flags apply to all format specifiers. The following sections explain each in detail.

**Justifying Output**

By default, all output is right-justified. That is, if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force output to be left-justified by placing a minus sign directly after the %. For instance, `%–10.2f` left-justifies a floating-point number with two decimal places in a 10-character field. For example, consider this program:

```java
// Demonstrate left justification.
import java.util.*;

class LeftJustify {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        // Right justify by default
        fmt.format("|%10.2f|", 123.123);
        System.out.println(fmt);
        fmt.close();

        // Now, left justify.
        fmt = new Formatter();
        fmt.format("|%-10.2f|", 123.123);
        System.out.println(fmt);
        fmt.close();
    }
}
```
It produces the following output:

|    123.12|
|123.12    |
|123.12    |

As you can see, the second line is left-justified within a 10-character field.

The Space, +, 0, and ( Flags

To cause a + sign to be shown before positive numeric values, add the + flag. For example,

```java
fmt.format("%+d", 100);
```

creates this string:

+100

When creating columns of numbers, it is sometimes useful to output a space before positive values so that positive and negative values line up. To do this, add the space flag. For example,

```java
// Demonstrate the space format specifiers.
import java.util.*;

class FormatDemo5 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("% d", -100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", 100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
```
fmt.format("% d", -200);
System.out.println(fmt);
fmt.close();

fmt = new Formatter();
fmt.format("% d", 200);
System.out.println(fmt);
fmt.close();
}
}

The output is shown here:

-100
100
-200
200

Notice that the positive values have a leading space, which causes the digits in the column to line up properly.

To show negative numeric output inside parentheses, rather than with a leading −, use the ( flag. For example,

fmt.format("%(d", -100);

creates this string:

(100)

The 0 flag causes output to be padded with zeros rather than spaces.

The Comma Flag

When displaying large numbers, it is often useful to add grouping separators, which in English are commas. For example, the value 1234567 is more easily read when formatted as 1,234,567. To add grouping specifiers, use
the comma (,) flag. For example,

```python
fmt.format("%,.2f", 4356783497.34);
```

creates this string:

```
4,356,783,497.34
```

The # Flag

The # can be applied to %o, %x, %e, and %f. For %e, and %f, the # ensures that there will be a decimal point even if there are no decimal digits. If you precede the %x format specifier with a #, the hexadecimal number will be printed with a 0x prefix. Preceding the %o specifier with # causes the number to be printed with a leading zero.

The Uppercase Option

As mentioned earlier, several of the format specifiers have uppercase versions that cause the conversion to use uppercase where appropriate. The following table describes the effect.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>%A</td>
<td>Causes the hexadecimal digits a through f to be displayed in uppercase as A through F. Also, the prefix 0x is displayed as 0X, and the p will be displayed as P.</td>
</tr>
<tr>
<td>%B</td>
<td>Uppercases the values true and false.</td>
</tr>
<tr>
<td>%E</td>
<td>Causes the e symbol that indicates the exponent to be displayed in uppercase.</td>
</tr>
<tr>
<td>%G</td>
<td>Causes the e symbol that indicates the exponent to be displayed in uppercase.</td>
</tr>
<tr>
<td>%H</td>
<td>Causes the hexadecimal digits a through f to be displayed in uppercase as A through F.</td>
</tr>
<tr>
<td>%S</td>
<td>Uppercases the corresponding string.</td>
</tr>
<tr>
<td>%T</td>
<td>Causes all alphabetical output to be displayed in uppercase.</td>
</tr>
<tr>
<td>%X</td>
<td>Causes the hexadecimal digits a through f to be displayed in uppercase as A through F. Also, the optional prefix 0x is displayed as 0X, if present.</td>
</tr>
</tbody>
</table>

For example, this call:
Using an Argument Index

Formatter includes a very useful feature that lets you specify the argument to which a format specifier applies. Normally, format specifiers and arguments are matched in order, from left to right. That is, the first format specifier matches the first argument, the second format specifier matches the second argument, and so on. However, by using an argument index, you can explicitly control which argument a format specifier matches.

An argument index immediately follows the % in a format specifier. It has the following format:

n$

where n is the index of the desired argument, beginning with 1. For example, consider this example:

fmt.format("%3$d %1$d %2$d", 10, 20, 30);

It produces this string:

30 10 20
In this example, the first format specifier matches 30, the second matches 10, and the third matches 20. Thus, the arguments are used in an order other than strictly left to right.

One advantage of argument indexes is that they enable you to reuse an argument without having to specify it twice. For example, consider this line:

```java
fmt.format("%d in hex is %1$x", 255);
```

It produces the following string:

```
255 in hex is ff
```

As you can see, the argument 255 is used by both format specifiers.

There is a convenient shorthand called a relative index that enables you to reuse the argument matched by the preceding format specifier. Simply specify `<` for the argument index. For example, the following call to `format()` produces the same results as the previous example:

```java
fmt.format("%d in hex is %<x", 255);
```

Relative indexes are especially useful when creating custom time and date formats. Consider the following example:

```java
// Use relative indexes to simplify the
// creation of a custom time and date format.
import java.util.*;

class FormatDemo6 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();
```
Here is sample output:

Today is day 1 of January, 2011

Because of relative indexing, the argument cal need only be passed once, rather than three times.

Closing a Formatter

In general, you should close a Formatter when you are done using it. Doing so frees any resources that it was using. This is especially important when formatting to a file, but it can be important in other cases, too. As the previous examples have shown, one way to close a Formatter is to explicitly call close(). However, beginning with JDK 7, Formatter implements the AutoCloseable interface. This means that it supports the new try-with-resources statement. Using this approach, the Formatter is automatically closed when it is no longer needed.

The try-with-resources statement is described in Chapter 13, in connection with files, because files are some of the most commonly used resources that must be closed. However, the same basic techniques apply here. For example, here is the first Formatter example reworked to use automatic resource management:

```java
// Use automatic resource management with Formatter.
import java.util.*;
```
class FormatDemo {
    public static void main(String args[]) {
        try (Formatter fmt = new Formatter()) {
            fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
            System.out.println(fmt);
        }
    }
}

The output is the same as before.

The Java printf( ) Connection

Although there is nothing technically wrong with using Formatter directly (as the preceding examples have done) when creating output that will be displayed on the console, there is a more convenient alternative: the printf( ) method. The printf( ) method automatically uses Formatter to create a formatted string. It then displays that string on System.out, which is the console by default. The printf( ) method is defined by both PrintStream and PrintWriter. The printf( ) method is described in Chapter 19.

Scanner

Scanner is the complement of Formatter. It reads formatted input and converts it into its binary form. Scanner can be used to read input from the console, a file, a string, or any source that implements theReadable interface orReadableByteChannel. For example, you can use Scanner to read a number from the keyboard and assign its value to a variable. As you will see, given its power, Scanner is surprisingly easy to use.
The Scanner Constructors

Scanner defines the constructors shown in Table 18-14. In general, a Scanner can be created for a String, an InputStream, a File, or any object that implements the Readable or ReadableByteChannel interfaces. Here are some examples.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanner(File from) throws FileNotFoundException</td>
<td>Creates a Scanner that uses the file specified by from as a source for input.</td>
</tr>
<tr>
<td>Scanner(File from, String charset) throws FileNotFoundException</td>
<td>Creates a Scanner that uses the file specified by from with the encoding specified by charset as a source for input.</td>
</tr>
<tr>
<td>Scanner(InputStream from)</td>
<td>Creates a Scanner that uses the stream specified by from as a source for input.</td>
</tr>
<tr>
<td>Scanner(InputStream from, String charset)</td>
<td>Creates a Scanner that uses the stream specified by from with the encoding specified by charset as a source for input.</td>
</tr>
<tr>
<td>Scanner(Path from) throws IOException</td>
<td>Creates a Scanner that uses the file specified by from as a source for input. (Added by JDK 7.)</td>
</tr>
<tr>
<td>Scanner(Path from, String charset) throws IOException</td>
<td>Creates a Scanner that uses the file specified by from with the encoding specified by charset as a source for input. (Added by JDK 7.)</td>
</tr>
<tr>
<td>Scanner(Readable from)</td>
<td>Creates a Scanner that uses the Readable object specified by from as a source for input.</td>
</tr>
<tr>
<td>Scanner(ReadableByteChannel from)</td>
<td>Creates a Scanner that uses the ReadableByteChannel specified by from as a source for input.</td>
</tr>
<tr>
<td>Scanner(ReadableByteChannel from, String charset)</td>
<td>Creates a Scanner that uses the ReadableByteChannel specified by from with the encoding specified by charset as a source for input.</td>
</tr>
<tr>
<td>Scanner(String from)</td>
<td>Creates a Scanner that uses the string specified by from as a source for input.</td>
</tr>
</tbody>
</table>

Table 18-14 The Scanner Constructors

The following sequence creates a Scanner that reads the file Test.txt:

```java
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

This works because FileReader implements the
Readable interface. Thus, the call to the constructor resolves to `Scanner(Readable)`.

This next line creates a `Scanner` that reads from standard input, which is the keyboard by default:

```java
Scanner conin = new Scanner(System.in);
```

This works because `System.in` is an object of type `InputStream`. Thus, the call to the constructor maps to `Scanner(InputStream)`.

The next sequence creates a `Scanner` that reads from a string.

```java
String instr = "10 99.88 scanning is easy.";
Scanner conin = new Scanner(instr);
```

Scanning Basics

Once you have created a `Scanner`, it is a simple matter to use it to read formatted input. In general, a `Scanner` reads tokens from the underlying source that you specified when the `Scanner` was created. As it relates to `Scanner`, a token is a portion of input that is delineated by a set of delimiters, which is whitespace by default. A token is read by matching it with a particular regular expression, which defines the format of the data. Although `Scanner` allows you to define the specific type of expression that its next input operation will match, it includes many predefined patterns, which match the primitive types, such as `int` and `double`, and strings. Thus, often you won’t need to specify a pattern to match.

In general, to use `Scanner`, follow this procedure:

1. Determine if a specific type of input is available
by calling one of Scanner’s hasNext X methods, where X is the type of data desired.

2. If input is available, read it by calling one of Scanner’s next X methods.

3. Repeat the process until input is exhausted.

4. Close the Scanner by calling close().

As the preceding indicates, Scanner defines two sets of methods that enable you to read input. The first are the hasNext X methods, which are shown in Table 18-15. These methods determine if the specified type of input is available. For example, calling hasNextInt() returns true only if the next token to be read is an integer. If the desired data is available, then you read it by calling one of Scanner’s next X methods, which are shown in Table 18-16. For example, to read the next integer, call nextInt(). The following sequence shows how to read a list of integers from the keyboard.

```java
Scanner conin = new Scanner(System.in);
int i;

// Read a list of integers.
while(conin.hasNextInt()) {
    i = conin.nextInt();
    // ...
}
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean hasNext()</td>
<td>Returns true if another token of any type is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNext(Pattern <em>pattern</em>)</td>
<td>Returns true if a token that matches the pattern passed in <em>pattern</em> is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNext(String <em>pattern</em>)</td>
<td>Returns true if a token that matches the pattern passed in <em>pattern</em> is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextBigDecimal()</td>
<td>Returns true if a value that can be stored in a BigDecimal object is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextBigInteger()</td>
<td>Returns true if a value that can be stored in a BigInteger object is available to be read. Returns false otherwise. The default radix is used. (Unless changed, the default radix is 10.)</td>
</tr>
<tr>
<td>boolean hasNextBigInteger(int <em>radix</em>)</td>
<td>Returns true if a value in the specified radix that can be stored in a BigInteger object is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextBoolean()</td>
<td>Returns true if a boolean value is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextByte()</td>
<td>Returns true if a byte value is available to be read. Returns false otherwise. The default radix is used. (Unless changed, the default radix is 10.)</td>
</tr>
<tr>
<td>boolean hasNextByte(int <em>radix</em>)</td>
<td>Returns true if a byte value in the specified radix is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextDouble()</td>
<td>Returns true if a double value is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextFloat()</td>
<td>Returns true if a float value is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextInt()</td>
<td>Returns true if an int value is available to be read. Returns false otherwise. The default radix is used. (Unless changed, the default radix is 10.)</td>
</tr>
<tr>
<td>boolean hasNextInt(int <em>radix</em>)</td>
<td>Returns true if an int value in the specified radix is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextLine()</td>
<td>Returns true if a line of input is available.</td>
</tr>
<tr>
<td>boolean hasNextLong()</td>
<td>Returns true if a long value is available to be read. Returns false otherwise. The default radix is used. (Unless changed, the default radix is 10.)</td>
</tr>
<tr>
<td>boolean hasNextLong(int <em>radix</em>)</td>
<td>Returns true if a long value in the specified radix is available to be read. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean hasNextShort()</td>
<td>Returns true if a short value is available to be read. Returns false otherwise. The default radix is used. (Unless changed, the default radix is 10.)</td>
</tr>
<tr>
<td>boolean hasNextShort(int <em>radix</em>)</td>
<td>Returns true if a short value in the specified radix is available to be read. Returns false otherwise.</td>
</tr>
</tbody>
</table>

Table 18-15 The Scanner hasNext Methods
The while loop stops as soon as the next token is not an integer. Thus, the loop stops reading integers as soon as a non-integer is encountered in the input stream.

If a next method cannot find the type of data it is looking for, it throws an InputMismatchException. A NoSuchElementException is thrown if no more input is
available. For this reason, it is best to first confirm that
the desired type of data is available by calling a hasNext
method before calling its corresponding next method.

Some Scanner Examples

Scanner makes what could be a tedious task into an easy
one. To understand why, let’s look at some examples. The
following program averages a list of numbers
entered at the keyboard:

```java
// Use Scanner to compute an average of the values.
import java.util.*;

class AvgNums {
    public static void main(String args[]) {
        Scanner conin = new Scanner(System.in);

        int count = 0;
        double sum = 0.0;

        System.out.println("Enter numbers to average.");

        // Read and sum numbers.
        while(conin.hasNext()) {
            if(conin.hasNextDouble()) {
                sum += conin.nextDouble();
                count++;
            } else {
                String str = conin.next();
                if(str.equals("done")) break;
                else {
                    System.out.println("Data format error.");
                    return;
                }
            }
        }

        conin.close();
        System.out.println("Average is "+ sum / count);
    }
}
```

The program reads numbers from the keyboard,
The program reads numbers from the keyboard, summing them in the process, until the user enters the string “done”. It then stops input and displays the average of the numbers. Here is a sample run:

Enter numbers to average.
1.2
2
3.4
4
done
Average is 2.65

The program reads numbers until it encounters a token that does not represent a valid double value. When this occurs, it confirms that the token is the string “done”. If it is, the program terminates normally. Otherwise, it displays an error.

Notice that the numbers are read by calling nextDouble( ). This method reads any number that can be converted into a double value, including an integer value, such as 2, and a floating-point value like 3.4. Thus, a number read by nextDouble( ) need not specify a decimal point. This same general principle applies to all next methods. They will match and read any data format that can represent the type of value being requested.

One thing that is especially nice about Scanner is that the same technique used to read from one source can be used to read from another. For example, here is the preceding program reworked to average a list of numbers contained in a text file:

```java
// Use Scanner to compute an average of the values in a file.
import java.util.*;
import java.io.*;

class AvgFile {
```
public static void main(String args[])
    throws IOException {

    int count = 0;
    double sum = 0.0;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Read and sum numbers.
    while(src.hasNext()) {
        if(src.hasNextDouble()) {
            sum += src.nextDouble();
            count++;
        } else {
            String str = src.next();
            if(str.equals("done")) break;
            else {
                System.out.println("File format error.");
                return;
            }
        }
    }

    src.close();
    System.out.println("Average is " + sum / count);
}
}

Here is the output:

Average is 6.2

The preceding program illustrates another important feature of Scanner. Notice that the file reader referred to by fin is not closed directly. Rather, it is closed automatically when src calls close( ). When you close a Scanner, the Readable associated with it is also closed (if
that Readable implements the Closeable interface. Therefore, in this case, the file referred to by fin is automatically closed when src is closed.

Beginning with JDK 7, Scanner also implements the AutoCloseable interface. This means that it can be managed by a try-with-resources block. As explained in Chapter 13, when try-with-resources is used, the scanner is automatically closed when the block ends. For example, src in the preceding program could have been managed like this:

```java
try (Scanner src = new Scanner(fin)) {
    // Read and sum numbers.
    while(src.hasNext()) {
        if(src.hasNextDouble()) {
            sum += src.nextDouble();
            count++;
        } else {
            String str = src.next();
            if(str.equals("done")) break;
            else {
                System.out.println("File format error.");
                return;
            }
        }
    }
}
```

To clearly demonstrate the closing of a Scanner, the following examples will call close() explicitly. Doing so also allows them to be compiled by versions of Java prior to JDK 7. However, the try-with-resources approach is more streamlined and can help prevent errors. Its use is recommended for new code.

One other point: To keep this and the other examples in this section compact, I/O exceptions are simply
thrown out of main(). However, your real-world code will normally handle I/O exceptions itself.

You can use Scanner to read input that contains several different types of data—even if the order of that data is unknown in advance. You must simply check what type of data is available before reading it. For example, consider this program:

```java
// Use Scanner to read various types of data from a file.
import java.util.*;
import java.io.*;

class ScanMixed {
    public static void main(String args[])
        throws IOException {

        int i;
        double d;
        boolean b;
        String str;
        // Write output to a file.
        FileWriter fout = new FileWriter("test.txt");
        fout.write("Testing Scanner 10 12.2 one true two false");
        fout.close();

        FileReader fin = new FileReader("Test.txt");
        Scanner src = new Scanner(fin);
```
Here is the output:

String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false

When reading mixed data types, as the preceding program does, you need to be a bit careful about the order in which you call the next methods. For example, if the loop reversed the order of the calls to `nextInt( )` and `nextDouble( )`, both numeric values would have been read as doubles, because `nextDouble( )` matches any numeric string that can be represented as a `double`.

Setting Delimiters
Scanner defines where a token starts and ends based on a set of delimiters. The default delimiters are the whitespace characters, and this is the delimiter set that the preceding examples have used. However, it is possible to change the delimiters by calling the useDelimiter( ) method, shown here:

```java
Scanner useDelimiter(String pattern)

Scanner useDelimiter(Pattern pattern)
```

Here, pattern is a regular expression that specifies the delimiter set.

Here is the program that reworks the average program shown earlier so that it reads a list of numbers that are separated by commas, and any number of spaces:

```java
// Use Scanner to compute an average a list of
// comma-separated values.
import java.util.*;
import java.io.*;

class SetDelimiters {
    public static void main(String args[])
        throws IOException {
        int count = 0;
        double sum = 0.0;

        // Write output to a file.
        FileWriter fout = new FileWriter("test.txt");

        // Now, store values in comma-separated list.
        fout.write("2, 3.4, 5, 6, 7.4, 9.1, 10.5, done");
        fout.close();

        FileReader fin = new FileReader("Test.txt");
```
In this version, the numbers written to test.txt are separated by commas and spaces. The use of the delimiter pattern ‘, * ’ tells Scanner to match a comma and zero or more spaces as delimiters. The output is the same as before.

You can obtain the current delimiter pattern by calling delimiter( ), shown here:

```
Pattern delimiter( )
```

**Other Scanner Features**

Scanner defines several other methods in addition to those already discussed. One that is particularly useful in some circumstances is findInLine( ). Its general forms are shown here:
String findInLine(Pattern pattern)
String findInLine(String pattern)

This method searches for the specified pattern within the next line of text. If the pattern is found, the matching token is consumed and returned. Otherwise, null is returned. It operates independently of any delimiter set. This method is useful if you want to locate a specific pattern. For example, the following program locates the Age field in the input string and then displays the age:

```java
// Demonstrate findInLine().
import java.util.*;

class FindInLineDemo {
    public static void main(String args[]) {
        String instr = "Name: Tom Age: 28 ID: 77";

        Scanner conin = new Scanner(instr);

        // Find and display age.
        conin.findInLine("Age:"); // find Age
        if(conin.hasNext())
            System.out.println(conin.next());
        else
            System.out.println("Error!");

        conin.close();
    }
}
```

The output is 28. In the program, findInLine() is used to find an occurrence of the pattern “Age”. Once found, the next token is read, which is the age.

Related to findInLine() is findWithinHorizon(). It is shown here:

```java
String findWithinHorizon(Pattern pattern, int count)
```
String findWithinHorizon(String pattern, int count)

This method attempts to find an occurrence of the specified pattern within the next count characters. If successful, it returns the matching pattern. Otherwise, it returns null. If count is zero, then all input is searched until either a match is found or the end of input is encountered.

You can bypass a pattern using skip(), shown here:

Scanner skip(Pattern pattern)

Scanner skip(String pattern)

If pattern is matched, skip() simply advances beyond it and returns a reference to the invoking object. If pattern is not found, skip() throws NoSuchElementException.

Other Scanner methods include radix(), which returns the default radix used by the Scanner; useRadix(), which sets the radix; reset(), which resets the scanner; and close(), which closes the scanner.

The ResourceBundle, ListResourceBundle, and PropertyResourceBundle Classes
The java.util package includes three classes that aid in the internationalization of your program. The first is the abstract class ResourceBundle. It defines methods that enable you to manage a collection of locale-sensitive resources, such as the strings that are used to label the user interface elements in your program. You can define two or more sets of translated strings that support various languages, such as English, German, or Chinese,
with each translation set residing in its own bundle. You can then load the bundle appropriate to the current locale and use the strings to construct the program’s user interface.

Resource bundles are identified by their family name (also called their base name). To the family name can be added a two-character lowercase language code which specifies the language. In this case, if a requested locale matches the language code, then that version of the resource bundle is used. For example, a resource bundle with a family name of SampleRB could have a German version called SampleRB_de and a Russian version called SampleRB_ru. (Notice that an underscore links the family name to the language code.) Therefore, if the locale is Locale.GERMAN, SampleRB_de will be used.

It is also possible to indicate specific variants of a language that relate to a specific country by specifying a country code after the language code. A country code is a two-character uppercase identifier, such as AU for Australia or IN for India. A country code is also preceded by an underscore when linked to the resource bundle name. A resource bundle that has only the family name is the default bundle. It is used when no language-specific bundles are applicable.

NOTE

The language codes are defined by ISO standard 639 and the country codes by ISO standard 3166.

The methods defined by ResourceBundle are
One important point: null keys are not allowed and several of the methods will throw a NullPointerException if null is passed as the key. Notice the nested class ResourceBundle.Control. It was added by Java SE 6 and is used to control the resource-bundle loading process.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static final void clearCache( )</td>
<td>Deletes all resource bundles from the cache that were loaded by the default class loader.</td>
</tr>
<tr>
<td>static final void clearCache(ClassLoader ldr)</td>
<td>Deletes all resource bundles from the cache that were loaded by ldr.</td>
</tr>
<tr>
<td>boolean containsKey(String k)</td>
<td>Returns true if k is a key within the invoking resource bundle (or its parent).</td>
</tr>
<tr>
<td>static final ResourceBundle getBundle(String familyName)</td>
<td>Loads the resource bundle with a family name of familyName using the default locale and the default class loader. Throws MissingResourceException if no resource bundle matching the family name specified by familyName is available.</td>
</tr>
<tr>
<td>static final ResourceBundle getBundle(String familyName, Locale loc)</td>
<td>Loads the resource bundle with a family name of familyName using the specified locale and the default class loader. Throws MissingResourceException if no resource bundle matching the family name specified by familyName is available.</td>
</tr>
<tr>
<td>static ResourceBundle getBundle(String familyName, Locale loc, ClassLoader ldr)</td>
<td>Loads the resource bundle with a family name of familyName using the specified locale and the specified class loader. Throws MissingResourceException if no resource bundle matching the family name specified by familyName is available.</td>
</tr>
<tr>
<td>static final ResourceBundle getBundle(String familyName, ResourceBundle.Control cntrl)</td>
<td>Loads the resource bundle with a family name of familyName using the default locale and the default class loader. The loading process is under the control of cntrl. Throws MissingResourceException if no resource bundle matching the family name specified by familyName is available.</td>
</tr>
</tbody>
</table>
Table 18-17 The Methods Defined by ResourceBundle

There are two subclasses of ResourceBundle. The first is PropertyResourceBundle, which manages resources by using property files. PropertyResourceBundle adds no methods of its own. The second is the abstract class ListResourceBundle, which manages resources in an array
of key/value pairs. ListResourceBundle adds the method
getContents( ), which all subclasses must implement. It is
shown here:

    protected abstract Object[][] getContents()

It returns a two-dimensional array that contains
key/value pairs that represent resources. The keys must
be strings. The values are typically strings, but can be
other types of objects.

Here is an example that demonstrates using a resource
bundle. The resource bundle has the family name
SampleRB. Two resource bundle classes of this family
are created by extending ListResourceBundle. The first is
called SampleRB, and it is the default bundle (which
uses English). It is shown here:

    import java.util.*;
    public class SampleRB extends ListResourceBundle {
        protected Object[][] getContents() {
            Object[][] resources = new Object[3][2];

            resources[0][0] = "title";
            resources[0][1] = "My Program";

            resources[1][0] = "StopText";
            resources[1][1] = "Stop";

            resources[2][0] = "StartText";
            resources[2][1] = "Start";

            return resources;
        }
    }

The second resource bundle, shown next, is called
SampleRB_de. It contains the German translation.

    import java.util.*;
The following program demonstrates these two resource bundles by displaying the string associated with each key for both the default (English) version and the German version:
class LRBDemo {
    public static void main(String[] args) {
        // Load the default bundle.
        ResourceBundle rd = ResourceBundle.getBundle("SampleRB");

        System.out.println("English version: ");
        System.out.println("String for Title key: " +
                rd.getString("title");
        System.out.println("String for StopText key: " +
                rd.getString("StopText");
        System.out.println("String for StartText key: " +
                rd.getString("StartText");

        // Load the German bundle.
        rd = ResourceBundle.getBundle("SampleRB", Locale.GERMAN);

        System.out.println("German version: ");
        System.out.println("String for Title key: " +
                rd.getString("title");
        System.out.println("String for StopText key: " +
                rd.getString("StopText");
        System.out.println("String for StartText key: " +
                rd.getString("StartText");
    }
}

The output from the program is shown here:

English version:
String for Title key: My Program
String for StopText key: Stop
String for StartText key: Start

German version:
String for Title key: Mein Programm
String for StopText key: Anschlag
String for StartText key: Anfang

Miscellaneous Utility Classes and Interfaces
In addition to the classes already discussed, java.util includes the following classes:
The following interfaces are also packaged in java.util:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventListener</td>
<td>Indicates that a class is an event listener. Events are discussed in Chapter 23.</td>
</tr>
<tr>
<td>Formattable</td>
<td>Enables a class to provide custom formatting.</td>
</tr>
</tbody>
</table>

The java.util Subpackages
Java defines the following subpackages to java.util:

- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks
- java.util.jar
- java.util.logging
- java.util.prefs
- java.util.regex
- java.util.spi
- java.util.zip

Each is briefly examined here.
subpackages, java.util.concurrent.atomic and java.util.concurrent.locks, support concurrent programming. These packages provide a high-performance alternative to using Java’s built-in synchronization features when thread-safe operation is required. Beginning with JDK 7, java.util.concurrent also provides the Fork/Join Framework. These packages are examined in detail in Chapter 27.

java.util.jar
The java.util.jar package provides the ability to read and write Java Archive (JAR) files.

java.util.logging
The java.util.logging package provides support for program activity logs, which can be used to record program actions, and to help find and debug problems.

java.util.prefs
The java.util.prefs package provides support for user preferences. It is typically used to support program configuration.

java.util.regex
The java.util.regex package provides support for regular expression handling. It is described in detail in Chapter 28.

java.util.spi
The java.util.spi package provides support for service providers.
The java.util.zip package provides the ability to read and write files in the popular ZIP and GZIP formats. Both ZIP and GZIP input and output streams are available.
This chapter explores java.io, which provides support for I/O operations. Chapter 13 presented an overview of Java’s I/O system, including basic techniques for reading and writing files, handling I/O exceptions, and closing a file. Here, we will examine the Java I/O system in greater detail.

As all programmers learn early on, most programs cannot accomplish their goals without accessing external data. Data is retrieved from an input source. The results of a program are sent to an output destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the stream. A stream, as explained in Chapter 13, is a logical entity that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ.

NOTE

The stream-based I/O system packaged in java.io and described in this chapter has been part of Java since its original release and is widely used. However, beginning with version 1.4, a second I/O
system was added to Java. It is called NIO (which was originally an acronym for New I/O). NIO is packaged in java.nio and its subpackages. With the release of JDK 7, the capabilities of the NIO system have been greatly expanded and its use is expected to grow. The NIO system is described in Chapter 20.

The I/O Classes and Interfaces

The I/O classes defined by java.io are listed here:

<table>
<thead>
<tr>
<th>Class</th>
<th>Class</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferedInputStream</td>
<td>FileWriter</td>
<td>PipedOutputStream</td>
</tr>
<tr>
<td>BufferedOutputStream</td>
<td>FilterInputStream</td>
<td>PipedReader</td>
</tr>
<tr>
<td>BufferedReader</td>
<td>FilterOutputStream</td>
<td>PipedWriter</td>
</tr>
<tr>
<td>BufferedWriter</td>
<td>FilterReader</td>
<td>PrintStream</td>
</tr>
<tr>
<td>ByteArrayInputStream</td>
<td>FilterWriter</td>
<td>PrintWriter</td>
</tr>
<tr>
<td>ByteArrayOutputStream</td>
<td>InputStreamReader</td>
<td>PushbackInputStream</td>
</tr>
<tr>
<td>CharArrayReader</td>
<td>InputStreamReader</td>
<td>PushbackReader</td>
</tr>
<tr>
<td>CharArrayWriter</td>
<td>LineNumberReader</td>
<td>RandomAccessFile</td>
</tr>
<tr>
<td>Console</td>
<td>ObjectInputStream</td>
<td>Reader</td>
</tr>
<tr>
<td>DataInputStream</td>
<td>ObjectInputStream.Field</td>
<td>SequenceInputStream</td>
</tr>
<tr>
<td>DataOutputStream</td>
<td>ObjectOutputStream</td>
<td>SerializablePermission</td>
</tr>
<tr>
<td>File</td>
<td>ObjectOutputStream.Field</td>
<td>StreamTokenizer</td>
</tr>
<tr>
<td>FileDescriptor</td>
<td>ObjectStreamClass</td>
<td>StringReader</td>
</tr>
<tr>
<td>FileInputStream</td>
<td>ObjectStreamField</td>
<td>StringWriter</td>
</tr>
<tr>
<td>FileOutputStream</td>
<td>OutputStream</td>
<td>Writer</td>
</tr>
<tr>
<td>FilePermission</td>
<td>OutputStreamWriter</td>
<td></td>
</tr>
<tr>
<td>FileReader</td>
<td>PipedInputStream</td>
<td></td>
</tr>
</tbody>
</table>

The java.io package also contains two deprecated classes that are not shown in the preceding table: LineNumberInputStream and StringBufferInputStream. These classes should not be used for new code.

The following interfaces are defined by java.io:
As you can see, there are many classes and interfaces in the `java.io` package. These include byte and character streams, and object serialization (the storage and retrieval of objects). This chapter examines several of the most commonly used I/O components. We begin our discussion with one of the most distinctive I/O classes: `File`.

**File**

Although most of the classes defined by `java.io` operate on streams, the `File` class does not. It deals directly with files and the file system. That is, the `File` class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A `File` object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

**NOTE**

The Path interface and Files class, added by JDK 7 to the NIO system, offer a powerful alternative to `File` in many cases. See Chapter 20 for details.

Files are a primary source and destination for data within many programs. Although there are severe
restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a File with one additional property—a list of filenames that can be examined by the list() method.

The following constructors can be used to create File objects:

- `File(String directoryPath)`
- `File(String directoryPath, String filename)`
- `File(File dirObj, String filename)`
- `File(URI uriObj)`

Here, `directoryPath` is the path name of the file; `filename` is the name of the file or subdirectory; `dirObj` is a File object that specifies a directory; and `uriObj` is a URI object that describes a file.

The following example creates three files: `f1`, `f2`, and `f3`. The first File object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to `f1` and a filename; `f3` refers to the same file as `f2`.

```java
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

**NOTE**

Java does the right thing with path separators
between UNIX and Windows conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly. Remember, if you are using the Windows convention of a backslash character (\), you will need to use its escape sequence (\\) within a string.

File defines many methods that obtain the standard properties of a File object. For example, getName( ) returns the name of the file; getParent( ) returns the name of the parent directory; and exists( ) returns true if the file exists, false if it does not. The File class, however, is not symmetrical. There are a few methods that allow you to examine the properties of a simple file object, but no corresponding function exists to change those attributes.

The following example demonstrates several of the File methods. It assumes that a directory called java exists off the root directory and that it contains a file called COPYRIGHT.

```java
// Demonstrate File.
import java.io.File;

class FileDemo {
    static void p(String s) {
        System.out.println(s);
    }

    public static void main(String args[]) {
        File f1 = new File("/java/COPYRIGHT");
        p("File Name: " + f1.getName());
        p("Path: " + f1.getPath());
        p("Abs Path: " + f1.getAbsolutePath());
    }

```
This program will produce output similar to this:

File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1282832030047
File size: 695 Bytes

Most of the File methods are self-explanatory. isFile() and isAbsolute() are not. isFile() returns true if called on a file and false if called on a directory. Also, isFile() returns false for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file. The isAbsolute() method returns true if the file has an absolute path and false if its path is relative.

File also includes two useful utility methods. The first is renameTo(), shown here:

boolean renameTo(File newName)
Here, the filename specified by newName becomes the new name of the invoking File object. It will return true upon success and false if the file cannot be renamed (if you attempt to rename a file so that it uses an existing filename, for example).

The second utility method is delete(), which deletes the disk file represented by the path of the invoking File object. It is shown here:

```java
boolean delete()
```

You can also use delete() to delete a directory if the directory is empty. delete() returns true if it deletes the file and false if the file cannot be removed.

Here are some other File methods that you will find helpful:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void deleteOnExit()</td>
<td>Removes the file associated with the invoking object when the Java Virtual Machine terminates.</td>
</tr>
<tr>
<td>long getFreeSpace()</td>
<td>Returns the number of free bytes of storage available on the partition associated with the invoking object.</td>
</tr>
<tr>
<td>long getTotalSpace()</td>
<td>Returns the storage capacity of the partition associated with the invoking object.</td>
</tr>
<tr>
<td>long getUsableSpace()</td>
<td>Returns the number of usable free bytes of storage available on the partition associated with the invoking object.</td>
</tr>
<tr>
<td>boolean isHidden()</td>
<td>Returns true if the invoking file is hidden. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean setLastModified(long millsec)</td>
<td>Sets the time stamp on the invoking file to that specified by millsec, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC).</td>
</tr>
<tr>
<td>boolean setReadOnly()</td>
<td>Sets the invoking file to read-only.</td>
</tr>
</tbody>
</table>

Methods also exist to mark files as readable, writable, and executable. Because File implements the
Comparable interface, the method compareTo( ) is also supported.

JDK 7 adds a new method to File called toPath( ), which is shown here:

    Path toPath( )

toPath( ) returns a Path object that represents the file encapsulated by the invoking File object. (In other words, toPath( ) converts a File into a Path.) Path is a new interface added by JDK 7. It is packaged in java.nio.file and is part of NIO. Thus, toPath( ) forms a bridge between the older File class and the new Path interface. (See Chapter 20 for a discussion of Path.)

Directories
A directory is a File that contains a list of other files and directories. When you create a File object that is a directory, the isDirectory( ) method will return true. In this case, you can call list( ) on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

    String[ ] list( )

The list of files is returned in an array of String objects.

The program shown here illustrates how to use list( ) to examine the contents of a directory:
Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory

Using FilenameFilter
You will often want to limit the number of files returned by the list() method to include only those files that...
match a certain filename pattern, or filter. To do this, you must use a second form of list( ), shown here:

   String[ ] list(FilenameFilter FFObj)

In this form, FFObj is an object of a class that implements the FilenameFilter interface.

FilenameFilter defines only a single method, accept( ), which is called once for each file in a list. Its general form is given here:

   boolean accept(File directory, String filename)

The accept( ) method returns true for files in the directory specified by directory that should be included in the list (that is, those that match the filename argument) and returns false for those files that should be excluded.

The OnlyExt class, shown next, implements FilenameFilter. It will be used to modify the preceding program to restrict the visibility of the filenames returned by list( ) to files with names that end in the file extension specified when the object is constructed.

   import java.io.*;

   public class OnlyExt implements FilenameFilter {

      String ext;

      public OnlyExt(String ext) {
         this.ext = "." + ext;
      }

      public boolean accept(File dir, String name) {
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);
        FilenameFilter only = new OnlyExt("html");
        String s[] = f1.list(only);

        for (int i=0; i < s.length; i++) {
            System.out.println(s[i]);
        }
    }
}

The listFiles() Alternative
There is a variation to the list( ) method, called listFiles( ), which you might find useful. The signatures for listFiles( ) are shown here:

    File[ ] listFiles( )
    File[ ] listFiles(FilenameFilter FObj)
    File[ ] listFiles(FileFilter FObj)

These methods return the file list as an array of File objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified FilenameFilter. Aside from returning an array
of File objects, these two versions of listFiles() work like their equivalent list() methods.

The third version of listFiles() returns those files with path names that satisfy the specified FileFilter. FileFilter defines only a single method, accept(), which is called once for each file in a list. Its general form is given here:

```java
boolean accept(File path)
```

The accept() method returns true for files that should be included in the list (that is, those that match the path argument) and false for those that should be excluded.

Creating Directories

Another two useful File utility methods are mkdir() and mkdirs(). The mkdir() method creates a directory, returning true on success and false on failure. Failure can occur for various reasons, such as the path specified in the File object already exists, or the directory cannot be created because the entire path does not exist yet. To create a directory for which no path exists, use the mkdirs() method. It creates both a directory and all the parents of the directory.

The AutoCloseable, Closeable, and Flushable Interfaces

There are three interfaces that are quite important to the stream classes. Two are Closeable and Flushable. They are defined in java.io and were added by JDK 5. The third, AutoCloseable, is new, being added by JDK 7. It is
AutoCloseable provides support for JDK 7’s new try-with-resources statement, which automates the process of closing a resource. (See Chapter 13.) Only objects of classes that implement AutoCloseable can be managed by try-with-resources. AutoCloseable is discussed in Chapter 16, but it is reviewed here for convenience. The AutoCloseable interface defines only the close() method:

```java
void close() throws Exception
```

This method closes the invoking object, releasing any resources that it may hold. It is called automatically at the end of a try-with-resources statement, thus eliminating the need to explicitly call close(). Because this interface is implemented by all of the I/O classes that open a stream, all such streams can be automatically closed by a try-with-resources statement. Automatically closing a stream ensures that it is properly closed when it is no longer needed, thus preventing memory leaks and other problems.

The Closeable interface also defines the close() method. Objects of a class that implement Closeable can be closed. Beginning with JDK 7, Closeable extends AutoCloseable. Therefore, in JDK 7, any class that implements Closeable also implements AutoCloseable.

Objects of a class that implements Flushable can force buffered output to be written to the stream to which the object is attached. It defines the flush() method, shown
void flush() throws IOException

Flushing a stream typically causes buffered output to be physically written to the underlying device. This interface is implemented by all of the I/O classes that write to a stream.

I/O Exceptions

Two exceptions play an important role in I/O handling. The first is IOException. As it relates to most of the I/O classes described in this chapter, if an I/O error occurs, an IOException is thrown. In many cases, if a file cannot be opened, a FileNotFoundException is thrown. FileNotFoundException is a subclass of IOException, so both can be caught with a single catch that catches IOException. For brevity, this is the approach used by most of the sample code in this chapter. However, in your own applications, you might find it useful to catch each exception separately.

Another exception class that is sometimes important when performing I/O is SecurityException. As explained in Chapter 13, in situations in which a security manager is present, several of the file classes will throw a SecurityException if a security violation occurs when attempting to open a file. By default, applications run via java do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a
Two Ways to Close a Stream

In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation. The techniques used to close a stream were described in Chapter 13, but because of their importance, they warrant a brief review here before the stream classes are examined.

Beginning with JDK 7, there are two basic ways in which you can close a stream. The first is to explicitly call close() on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, close() is typically called within a finally block. Thus, a simplified skeleton for the traditional approach is shown here:

```java
try {
    // open and access file
} catch (I/O-exception) {
    // ...
} finally {
    // close the file
}
```
This general technique (or variation thereof) is common in code that predates JDK 7.

The second approach to closing a stream is to automate the process by using the new try-with-resources statement that was added by JDK 7. The try-with-resources statement is an enhanced form of try that has the following form:

```java
try (resource-specification) {
   // use the resource
}
```

Here, resource-specification is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the try block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call close( ) explicitly.

Here are three key points about the try-with-resources statement:

- Resources managed by try-with-resources must be objects of classes that implement AutoCloseable.
- The resource declared in the try is implicitly final.
- You can manage more than one resource by separating each declaration by a semicolon.

Also, remember that the scope of the declared resource is
limited to the try-with-resources statement.

The principal advantage of try-with-resources is that the resource (in this case, a stream) is closed automatically when the try block ends. Thus, it is not possible to forget to close the stream, for example. The try-with-resources approach also typically results in shorter, clearer, easier-to-maintain source code.

Because of its advantages, try-with-resources is expected to be used extensively in new code. As a result, most of the code in this chapter (and in this book) will use it. However, because there are millions of lines of pre-JDK 7 code, it is important for all programmers to also be familiar with the traditional approach to closing a stream. For example, you will quite likely have to work on legacy code that uses the traditional approach or in an environment that uses a version of Java that predates JDK 7. There may also be times when the automated approach is not appropriate because of other aspects of your code. For this reason, a few I/O examples in this book will demonstrate the traditional approach so you can see it in action.

One last point: The examples that use try-with-resources must be compiled by a JDK 7 or later. They won’t work with an older compiler. The examples that use the traditional approach can be compiled by older versions of Java.

REMEMBER
Because try-with-resources streamlines the process of releasing a resource and eliminates the possibility of accidentally forgetting to release a resource, it is the approach recommended for new code when its use is appropriate.

The Stream Classes
Java’s stream-based I/O is built upon four abstract classes: InputStream, OutputStream, Reader, and Writer. These classes were briefly discussed in Chapter 13. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

InputStream and OutputStream are designed for byte streams. Reader and Writer are designed for character streams. The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings and use the byte stream classes when working with bytes or other binary objects.

In the remainder of this chapter, both the byte- and character-oriented streams are examined.

The Byte Streams
The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types
of programs. Since the byte stream classes are topped by InputStream and OutputStream, our discussion begins with them.

InputStream

InputStream is an abstract class that defines Java’s model of streaming byte input. It implements the AutoCloseable and Closeable interfaces. Most of the methods in this class will throw an IOException when an I/O error occurs. (The exceptions are mark( ) and markSupported( ).) Table 19-1 shows the methods in InputStream.

NOTE

Most of the methods described in Table 19-1 are implemented by the subclasses of InputStream. The mark( ) and reset( ) methods are exceptions; notice their use, or lack thereof, by each subclass in the discussions that follow.

OutputStream

OutputStream is an abstract class that defines streaming byte output. It implements the AutoCloseable, Closeable, and Flushable interfaces. Most of the methods in this class return void and throw an IOException in the case of I/O errors. Table 19-2 shows the methods in OutputStream.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int available( )</td>
<td>Returns the number of bytes of input currently available for reading.</td>
</tr>
<tr>
<td>void close( )</td>
<td>Closes the input source. Further read attempts will generate an IOException.</td>
</tr>
<tr>
<td>void mark(int numBytes)</td>
<td>Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.</td>
</tr>
<tr>
<td>boolean markSupported( )</td>
<td>Returns true if mark() / reset() are supported by the invoking stream.</td>
</tr>
<tr>
<td>int read( )</td>
<td>Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>int read(byte buffer[ ])</td>
<td>Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>int read(byte buffer[ ], int offset, int numBytes)</td>
<td>Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. -1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>void reset( )</td>
<td>Resets the input pointer to the previously set mark.</td>
</tr>
<tr>
<td>long skip(long numBytes)</td>
<td>Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.</td>
</tr>
</tbody>
</table>

Table 19-1 The Methods Defined by InputStream

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close( )</td>
<td>Closes the output stream. Further write attempts will generate an IOException.</td>
</tr>
<tr>
<td>void flush( )</td>
<td>Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.</td>
</tr>
<tr>
<td>void write(int b)</td>
<td>Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write( ) with an expression without having to cast it back to byte.</td>
</tr>
<tr>
<td>void write(byte buffer[ ])</td>
<td>Writes a complete array of bytes to an output stream.</td>
</tr>
<tr>
<td>void write(byte buffer[ ], int offset, int numBytes)</td>
<td>Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].</td>
</tr>
</tbody>
</table>

Table 19-2 The Methods Defined by OutputStream

**FileInputStream**

The FileInputStream class creates an InputStream that you can use to read bytes from a file. Two commonly
used constructors are shown here:

    FileInputStream(String filePath)
    FileInputStream(File fileObj)

Either can throw a FileNotFoundException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file.

The following example creates two FileInputStreams that use the same disk file and each of the two constructors:

    FileInputStream f0 = new FileInputStream("/autoexec.bat")
    File f = new File("/autoexec.bat");
    FileInputStream f1 = new FileInputStream(f);

Although the first constructor is probably more commonly used, the second allows you to closely examine the file using the File methods, before attaching it to an input stream. When a FileInputStream is created, it is also opened for reading. FileInputStream overrides six of the methods in the abstract class InputStream. The mark( ) and reset( ) methods are not overridden, and any attempt to use reset( ) on a FileInputStream will generate an IOException.

The next example shows how to read a single byte, an array of bytes, and a subrange of an array of bytes. It also illustrates how to use available( ) to determine the number of bytes remaining and how to use the skip( ) method to skip over unwanted bytes. The program reads its own source file, which must be in the current
// Demonstrate FileInputStream.  
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileInputStreamDemo {
    public static void main(String args[]) {
        int size;

        // Use try-with-resources to close the stream.
        try (FileInputStream f =
            new FileInputStream("FileInputStreamDemo.java") ) {

            System.out.println("Total Available Bytes: " +
                (size = f.available()));

            int n = size/40;
            System.out.println("First " + n +
                " bytes of the file one read() at a time");
            for (int i=0; i < n; i++) {
                System.out.print((char) f.read());
            }

            System.out.println("\nStill Available: " + f.available());

            System.out.println("Reading the next " + n +
                " with one read(b[])");
            byte b[] = new byte[n];
            if (f.read(b) != n) {
                System.err.println("couldn’t read " + n + " bytes.");
            }
        }
    }
}
Here is the output produced by this program:

```
Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
// Demonstrate FileInputStream.
// This pr
Still Available: 1741
Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J
Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
    if (  
Still Available: 827
```

This somewhat contrived example demonstrates how to read three ways, to skip input, and to inspect the amount of data available on a stream.

NOTE

The preceding example and the other examples in
this chapter handle any I/O exceptions that might occur as described in Chapter 13. See Chapter 13 for details and alternatives.

FileOutputStream

FileOutputStream creates an OutputStream that you can use to write bytes to a file. It implements the AutoCloseable, Closeable, and Flushable interfaces. Four of its constructors are shown here:

```java
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)
```

They can throw a FileNotFoundException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, the file is opened in append mode.

Creation of a FileOutputStream is not dependent on the file already existing. FileOutputStream will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an exception will be thrown.

The following example creates a sample buffer of bytes by first making a String and then using the `getBytes()` method to extract the byte array equivalent. It then creates three files. The first, file1.txt, will contain every other byte from the sample. The second, file2.txt,
will contain the entire set of bytes. The third and last, file3.txt, will contain only the last quarter.

// Demonstrate FileOutputStream.
// This program uses the traditional approach to closing a file.

import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n" + " to come to the aid of their country\n" + " and pay their due taxes."
        byte buf[] = source.getBytes();
        FileOutputStream f0 = null;
        FileOutputStream f1 = null;
        FileOutputStream f2 = null;

        try {
            f0 = new FileOutputStream("file1.txt");
            f1 = new FileOutputStream("file2.txt");
            f2 = new FileOutputStream("file3.txt");

            // write to first file
            for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

            // write to second file
            f1.write(buf);
        }
    }
}
Here are the contents of each file after running this program. First, file1.txt:

```
Nwi h iebralgo e
t oet h i ftercuty n a hi u ae.
```

Next, file2.txt:

```
Now is the time for all good men
to come to the aid of their country
and pay their due taxes.
```

Finally, file3.txt:

```
nd pay their due taxes.
```

As the comment at the top of the program states, the preceding program shows an example that uses the traditional approach to closing a file when it is no longer
needed. This approach is required by all versions of Java prior to JDK 7 and is widely used in legacy code. As you can see, quite a bit of rather awkward code is required to explicitly call close() because each call could generate an IOException if the close operation fails. This program can be substantially improved by using the new try-with-resources statement. For comparison, here is the revised version. Notice that it is much shorter and streamlined:

```java
// Demonstrate FileOutputStream.
// This version uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n" + " to come to the aid of their country\n" + " and pay their due taxes.";
        byte buf[] = source.getBytes();

        // Use try-with-resources to close the files.
        try (FileOutputStream f0 = new FileOutputStream("file1.txt"));
             FileOutputStream f1 = new FileOutputStream("file2.txt"));
             FileOutputStream f2 = new FileOutputStream("file3.txt") ) {

            // write to first file
            for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

            // write to second file
            f1.write(buf);

            // write to third file
            f2.write(buf, buf.length-buf.length/4, buf.length/4);
        } catch (IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}
ByteArrayInputStream

ByteArrayInputStream is an implementation of an input stream that uses a byte array as the source. This class has two constructors, each of which requires a byte array to provide the data source:

```
ByteArrayInputStream(byte array [])
ByteArrayInputStream(byte array [], int start, int numBytes)
```

Here, array is the input source. The second constructor creates an InputStream from a subset of the byte array that begins with the character at the index specified by start and is numBytes long.

The close( ) method has no effect on a ByteArrayInputStream. Therefore, it is not necessary to call close( ) on a ByteArrayInputStream, but doing so is not an error.

The following example creates a pair of ByteArrayInputStreams, initializing them with the byte representation of the alphabet:

```
// Demonstrate ByteArrayInputStream.
import java.io.*;

class ByteArrayInputStreamDemo {
    public static void main(String args[]) {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        byte b[] = tmp.getBytes();
        ByteArrayInputStream input1 = new ByteArrayInputStream(b);
        ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0, 3);
    }
}
```
The input1 object contains the entire lowercase alphabet, whereas input2 contains only the first three letters.

A ByteArrayInputStream implements both mark( ) and reset( ). However, if mark( ) has not been called, then reset( ) sets the stream pointer to the start of the stream—which, in this case, is the start of the byte array passed to the constructor. The next example shows how to use the reset( ) method to read the same input twice. In this case, the program reads and prints the letters “abc” once in lowercase and then again in uppercase.

```java
import java.io.*;

class ByteArrayInputStreamReset {
    public static void main(String args[]) {
        String tmp = "abc";
        byte b[] = tmp.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(b);

        for (int i=0; i<2; i++) {
            int c;
            while ((c = in.read()) != -1) {
                if (i == 0) {
                    System.out.print((char) c);
                } else {
                    System.out.print(Character.toUpperCase((char) c));
                }
            }
            System.out.println();
            in.reset();
        }
    }
}
```

This example first reads each character from the stream and prints it as-is in lowercase. It then resets the stream
ByteArrayOutputStream

ByteArrayOutputStream is an implementation of an output stream that uses a byte array as the destination. ByteArrayOutputStream has two constructors, shown here:

    ByteArrayOutputStream()  
    ByteArrayOutputStream(int numBytes)

In the first form, a buffer of 32 bytes is created. In the second, a buffer is created with a size equal to that specified by numBytes. The buffer is held in the protected buf field of ByteArrayOutputStream. The buffer size will be increased automatically, if needed. The number of bytes held by the buffer is contained in the protected count field of ByteArrayOutputStream.

The close() method has no effect on a ByteArrayOutputStream. Therefore, it is not necessary to call close() on a ByteArrayOutputStream, but doing so is not an error.

The following example demonstrates ByteArrayOutputStream:
When you run the program, you will create the following output. Notice how after the call to reset( ),

```java
// Demonstrate ByteArrayInputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class ByteArrayInputStreamDemo {
    public static void main(String args[]) {
        ByteArrayInputStream f = new ByteArrayInputStream(new String("This should end up in the array").getBytes());
        try {
            f.write(buf);
        } catch( IOException e) {
            System.out.println("Error Writing to Buffer");
            return;
        }

        System.out.println("Buffer as a string");
        System.out.println(f.toString());
        System.out.println("Into array");
        byte b[] = f.toByteArray();
        for (int i=0; i<b.length; i++) System.out.print((char) b[i]);
        System.out.println("\nTo an OutputStream()" );

        // Use try-with-resources to manage the file stream.
        try { FileOutputStream f2 = new FileOutputStream("test.txt")
        { f.writeTo(f2);
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
            return;
        }
        System.out.println("Doing a reset");
        f.reset();
        for (int i=0; i<3; i++) f.write('X');
        System.out.println(f.toString());
    }
}
```
Buffer as a string  
This should end up in the array  
Into array  
This should end up in the array  
To an OutputStream()  
Doing a reset  
XXX

This example uses the writeTo( ) convenience method to write the contents of f to test.txt. Examining the contents of the test.txt file created in the preceding example shows the result we expected:

This should end up in the array

Filtered Byte Streams

Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. These streams are typically accessed by methods that are expecting a generic stream, which is a superclass of the filtered streams. Typical extensions are buffering, character translation, and raw data translation. The filtered byte streams are FilterInputStream and FilterOutputStream. Their constructors are shown here:

FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)

The methods provided in these classes are identical to those in InputStream and OutputStream.
Buffered Byte Streams

For the byte-oriented streams, a buffered stream extends a filtered stream class by attaching a memory buffer to the I/O stream. This buffer allows Java to do I/O operations on more than a byte at a time, thereby improving performance. Because the buffer is available, skipping, marking, and resetting of the stream become possible. The buffered byte stream classes are BufferedInputStream and BufferedOutputStream. PushbackInputStream also implements a buffered stream.

BufferedInputStream

Buffering I/O is a very common performance optimization. Java’s BufferedInputStream class allows you to “wrap” any InputStream into a buffered stream to improve performance.

BufferedInputStream has two constructors:

BufferedInputStream(InputStream inputStream)
BufferedInputStream(InputStream inputStream, int bufSize)

The first form creates a buffered stream using a default buffer size. In the second, the size of the buffer is passed in bufSize. Use of sizes that are multiples of a memory page, a disk block, and so on, can have a significant positive impact on performance. This is, however, implementation-dependent. An optimal buffer size is generally dependent on the host operating system, the
amount of memory available, and how the machine is configured. To make good use of buffering doesn’t necessarily require quite this degree of sophistication. A good guess for a size is around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the InputStream, you will be manipulating fast memory most of the time.

Buffering an input stream also provides the foundation required to support moving backward in the stream of the available buffer. Beyond the read( ) and skip( ) methods implemented in any InputStream, BufferedInputStream also supports the mark( ) and reset( ) methods. This support is reflected by BufferedInputStream.markSupported( ) returning true.

The following example contrives a situation where we can use mark( ) to remember where we are in an input stream and later use reset( ) to get back there. This example is parsing a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the reset( ) happens and where it does not.
import java.io.*;

class BufferedInputStreamDemo {
    public static void main(String args[]) {
        String s = "This is a © copyright symbol " +
                   "but this is © not.\n";
        byte buf[] = s.getBytes();

        ByteArrayInputStream in = new ByteArrayInputStream(buf);
        int c;
        boolean marked = false;

        // Use try-with-resources to manage the file.
        try (BufferedInputStream f = new BufferedInputStream(in)) {
            while ((c = f.read()) != -1) {
                switch(c) {
                    case '©':
                        if (!marked) {
                            f.mark(32);
                            marked = true;
                        } else {
                            marked = false;
                        }
                        break;
                    case ';':
                        if (marked) {
                            marked = false;
                            System.out.print("(c)"s);
                        } else

Notice that this example uses mark(32), which preserves the mark for the next 32 bytes read (which is enough for all entity references). Here is the output produced by this program:

```
This is a (c) copyright symbol but this is &copy not.
```

**BufferedOutputStream**

A BufferedOutputStream is similar to any OutputStream with the exception of an added flush( ) method that is used to ensure that data buffers are written to the stream being buffered. Since the point of a BufferedOutputStream is to improve performance by reducing the number of times the system actually writes data, you may need to call flush( ) to cause any data that is in the buffer to be immediately written.

Unlike buffered input, buffering output does not
Buffers for output in Java are there to increase performance. Here are the two available constructors:

```
BufferedOutputStream(OutputStream outputStream)
BufferedOutputStream(OutputStream outputStream, int bufSize)
```

The first form creates a buffered stream using the default buffer size. In the second form, the size of the buffer is passed in `bufSize`.

### PushbackInputStream

One of the novel uses of buffering is the implementation of pushback. Pushback is used on an input stream to allow a byte to be read and then returned (that is, “pushed back”) to the stream. The `PushbackInputStream` class implements this idea. It provides a mechanism to “peek” at what is coming from an input stream without disrupting it.

```
PushbackInputStream(InputStream inputStream)
PushbackInputStream(InputStream inputStream, int numBytes)
```

The first form creates a stream object that allows one byte to be returned to the input stream. The second form creates a stream that has a pushback buffer that is `numBytes` long. This allows multiple bytes to be returned
to the input stream.

Beyond the familiar methods of InputStream, PushbackInputStream provides unread( ), shown here:

```java
void unread(int b)
void unread(byte buffer [])
void unread(byte buffer, int offset, int numBytes)
```

The first form pushes back the low-order byte of b. This will be the next byte returned by a subsequent call to read( ). The second form pushes back the bytes in buffer. The third form pushes back numBytes bytes beginning at offset from buffer. An IOException will be thrown if there is an attempt to push back a byte when the pushback buffer is full.

Here is an example that shows how a programming language parser might use a PushbackInputStream and unread( ) to deal with the difference between the = operator for comparison and the = operator for assignment:
Here is the output for this example. Notice that \(=\) was replaced by "\(\text{.eq.}\)" and \(=\) was replaced by "\(<-\)".

```
if (a .eq. 4) a <- 0;
```

**CAUTION**

`PushbackInputStream` has the side effect of invalidating the `mark()` or `reset()` methods of the `InputStream` used to create it. Use `markSupported()`
SequenceInputStream

The SequenceInputStream class allows you to concatenate multiple InputStreams. The construction of a SequenceInputStream is different from any other InputStream. A SequenceInputStream constructor uses either a pair of InputStreams or an Enumeration of InputStreams as its argument:

```
    SequenceInputStream(InputStream first, InputStream second)
    SequenceInputStream(Enumeration <? extends InputStream> streamEnum)
```

Operationally, the class fulfills read requests from the first InputStream until it runs out and then switches over to the second one. In the case of an Enumeration, it will continue through all of the InputStreams until the end of the last one is reached. When the end of each file is reached, its associated stream is closed. Closing the stream created by SequenceInputStream causes all unclosed streams to be closed.

Here is a simple example that uses a SequenceInputStream to output the contents of two files. For demonstration purposes, this program uses the traditional technique used to close a file. As an exercise, you might want to try changing it to use the try-with-

// Demonstrate sequenced input.
// This program uses the traditional approach to closing a file.

import java.io.*;
import java.util.*;

class InputStreamEnumerator implements Enumeration<FileInputStream> {
    private Enumeration<String> files;

    public InputStreamEnumerator(Vector<String> files) {
        this.files = files.elements();
    }

    public boolean hasMoreElements() {
        return files.hasMoreElements();
    }

    public FileInputStream nextElement() {
        try {
            return new FileInputStream(files.nextElement().toString());
        } catch (IOException e) {
            return null;
        }
    }
}
This example creates a Vector and then adds three filenames to it. It passes that vector of names to the InputStreamEnumerator class, which is designed to provide a wrapper on the vector where the elements returned are not the filenames but, rather, open FileInputStreams on those names. The SequenceInputStream opens each file in turn, and this example prints the contents of the files.

Notice in nextElement( ) that if a file cannot be opened, null is returned. This results in a NullPointerException, which is caught in main( ).
PrintStream class provides all of the output capabilities we have been using from the System file handle, System.out, since the beginning of the book. This makes PrintStream one of Java’s most often used classes. It implements the Appendable, AutoCloseable, Closeable, and Flushable interfaces.

PrintStream defines several constructors. The ones shown next have been specified from the start:

```java
PrintStream(OutputStream outputStream)
PrintStream(OutputStream outputStream, boolean flushOnNewline)
PrintStream(OutputStream outputStream, boolean flushOnNewline, String charset)
    throws UnsupportedEncodingException
```

Here, outputStream specifies an open OutputStream that will receive output. The flushOnNewline parameter controls whether the output buffer is automatically flushed every time a newline (\n) character or a byte array is written or when println() is called. If flushOnNewline is true, flushing automatically takes place. If it is false, flushing is not automatic. The first constructor does not automatically flush. You can specify a character encoding by passing its name in charset.

The next set of constructors gives you an easy way to construct a PrintStream that writes its output to a file:
These allow a PrintStream to be created from a File object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the PrintStream object directs all output to the specified file. You can specify a character encoding by passing its name in charSet.

NOTE

If a security manager is present, some PrintStream constructors will throw a SecurityException if a security violation occurs.

PrintStream supports the print( ) and println( ) methods for all types, including Object. If an argument is not a primitive type, the PrintStream methods will call the object’s toString( ) method and then display the result.
Somewhat recently (with the release of JDK 5), the printf() method was added to PrintStream. It allows you to specify the precise format of the data to be written. The printf() method uses the Formatter class (described in Chapter 18) to format data. It then writes this data to the invoking stream. Although formatting can be done manually, by using Formatter directly, printf() streamlines the process. It also parallels the C/C++ printf() function, which makes it easy to convert existing C/C++ code into Java. Frankly, printf() was a much welcome addition to the Java API because it greatly simplified the output of formatted data to the console.

The printf() method has the following general forms:

```java
PrintStream printf(String fmtString, Object ... args)
PrintStream printf(Locale loc, String fmtString, Object ... args)
```

The first version writes args to standard output in the format specified by fmtString, using the default locale. The second lets you specify a locale. Both return the invoking PrintStream.

In general, printf() works in a manner similar to the format() method specified by Formatter. The fmtString consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains format specifiers that define the way the subsequent arguments, specified by args, are displayed. For complete information on formatting
output, including a description of the format specifiers, see the Formatter class in Chapter 18.

Because System.out is a PrintStream, you can call printf() on System.out. Thus, printf() can be used in place of println() when writing to the console whenever formatted output is desired. For example, the following program uses printf() to output numeric values in various formats. Prior to JDK 5, such formatting required a bit of work. With the addition of printf(), this is now an easy task.

// Demonstrate printf().

class PrintfDemo {
    public static void main(String args[]) {
        System.out.println("Here are some numeric values " +
            "in different formats.\n");

        System.out.printf("Various integer formats: ");
        System.out.printf("%d %d %05d\n", 3, -3, 3, 3);

        System.out.println();
        System.out.printf("Default floating-point format: %f\n",
            1234567.123);
        System.out.printf("Floating-point with commas: %,f\n",
            1234567.123);
        System.out.printf("Negative floating-point default: %,f\n",
            -1234567.123);
        System.out.printf("Negative floating-point option: %,(f\n",
            -1234567.123);

        System.out.println();

        System.out.printf("Line up positive and negative values:\n");
        System.out.printf("% .2f% .2f\n",
            1234567.123, -1234567.123);
    }
}

The output is shown here:
Here are some numeric values in different formats.

Various integer formats: 3 (3) +3 00003

Default floating-point format: 1234567.123000
Floating-point with commas: 1,234,567.123000
Negative floating-point default: −1,234,567.123000
Negative floating-point option: (1,234,567.123000)

Line up positive and negative values:
   1,234,567.12
   −1,234,567.12

PrintStream also defines the format( ) method. It has these general forms:

PrintStream format(String fmtString, Object ... args)
PrintStream format(Locale loc, String fmtString, Object ... args)

It works exactly like printf( ).

DataOutputStream and DataInputStream

DataOutputStream and DataInputStream enable you to write or read primitive data to or from a stream. They implement the DataOutput and DataInput interfaces, respectively. These interfaces define methods that convert primitive values to or from a sequence of bytes. These streams make it easy to store binary data, such as integers or floating-point values, in a file. Each is examined here.

DataOutputStream extends FilterOutputStream, which extends OutputStream. In addition to implementing DataOutput, DataOutputStream also implements
DataOutputStream defines the following constructor:

    DataOutputStream(OutputStream outputStream)

Here, outputStream specifies the output stream to which data will be written. When a DataOutputStream is closed (by calling close()), the underlying stream specified by outputStream is also closed automatically.

DataOutputStream supports all of the methods defined by its superclasses. However, it is the methods defined by the DataOutput interface, which it implements, that make it interesting. DataOutput defines methods that convert values of a primitive type into a byte sequence and then writes it to the underlying stream. Here is a sampling of these methods:

    final void writeDouble(double value) throws IOException
    final void writeBoolean(boolean value) throws IOException
    final void writeInt(int value) throws IOException

Here, value is the value written to the stream.

DataInputStream is the complement of DataOutputStream. It extends FilterInputStream, which extends InputStream. In addition to implementing the DataInput interface, DataInputStream also implements AutoCloseable and Closeable. Here is its only constructor:
DataInputStream(InputStream inputStream)

Here, inputStream specifies the input stream from which data will be read. When a DataInputStream is closed (by calling close()), the underlying stream specified by inputStream is also closed automatically.

Like DataOutputStream, DataInputStream supports all of the methods of its superclasses, but it is the methods defined by the DataInput interface that make it unique. These methods read a sequence of bytes and convert them into values of a primitive type. Here is a sampling of these methods:

- final double readDouble() throws IOException
- final boolean readBoolean() throws IOException
- final int readInt() throws IOException

The following program demonstrates the use of DataOutputStream and DataInputStream:
import java.io.*;

class DataIODemo {
    public static void main(String args[]) throws IOException {
        // First, write the data.
        try (DataOutputStream dout =
             new DataOutputStream(new FileOutputStream("Test.dat"))) {
            dout.writeDouble(98.6);
            dout.writeInt(1000);
            dout.writeBoolean(true);
        } catch (FileNotFoundException e) {
            System.out.println("Cannot Open Output File");
            return;
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }

        // Now, read the data back.
        try (DataInputStream din =
             new DataInputStream(new FileInputStream("Test.dat"))) {
            double d = din.readDouble();
            int i = din.readInt();
            boolean b = din.readBoolean();

            System.out.println("Here are the values: "+d + " " + i + " " + b);
        } catch (FileNotFoundException e) {
            System.out.println("Cannot Open Input File");
            return;
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}

The output is shown here:

Here are the values: 98.6 1000 true

RandomAccessFile
RandomAccessFile encapsulates a random-access file. It is not derived from InputStream or OutputStream. Instead, it implements the interfaces DataInput and DataOutput, which define the basic I/O methods. It also implements the AutoCloseable and Closeable interfaces.

RandomAccessFile is special because it supports positioning requests—that is, you can position the file pointer within the file. It has these two constructors:

```java
RandomAccessFile(File fileObj, String access)
    throws FileNotFoundException

RandomAccessFile(String filename, String access)
    throws FileNotFoundException
```

In the first form, fileObj specifies the file to open as a File object. In the second form, the name of the file is passed in filename. In both cases, access determines what type of file access is permitted. If it is “r”, then the file can be read, but not written. If it is “rw”, then the file is opened in read-write mode. If it is “rws”, the file is opened for read-write operations and every change to the file’s data or metadata will be immediately written to the physical device. If it is “rwd”, the file is opened for read-write operations and every change to the file’s data will be immediately written to the physical device.

The method `seek()` shown here, is used to set the current position of the file pointer within the file:

```java
void seek(long newPos) throws IOException
```
Here, newPos specifies the new position, in bytes, of the file pointer from the beginning of the file. After a call to seek( ), the next read or write operation will occur at the new file position.

RandomAccessFile implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods. One is setLength( ). It has this signature:

```
void setLength(long len) throws IOException
```

This method sets the length of the invoking file to that specified by len. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

The Character Streams

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the “write once, run anywhere” philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the Reader and Writer abstract classes. We will begin with them.

NOTE
As discussed in Chapter 13, the character I/O classes were added by the 1.1 release of Java. Because of this, you may still find legacy code that uses byte streams where character streams would be more appropriate. When working on such code, it is a good idea to update it.

Reader
Reader is an abstract class that defines Java’s model of streaming character input. It implements the AutoCloseable, Closeable, and Readable interfaces. All of the methods in this class (except for markSupported( )) will throw an IOException on error conditions. Table 19-3 provides a synopsis of the methods in Reader.

Writer
Writer is an abstract class that defines streaming character output. It implements the AutoCloseable, Closeable, Flushable, and Appendable interfaces. All of the methods in this class throw an IOException in the case of errors. Table 19-4 shows a synopsis of the methods in Writer.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract void close( )</td>
<td>Closes the input source. Further read attempts will generate an IOException.</td>
</tr>
<tr>
<td>void mark(int numChars)</td>
<td>Places a mark at the current point in the input stream that will remain valid until numChars characters are read.</td>
</tr>
<tr>
<td>boolean markSupported( )</td>
<td>Returns true if mark( )/reset( ) are supported on this stream.</td>
</tr>
<tr>
<td>int read( )</td>
<td>Returns an integer representation of the next available character from the invoking input stream. −1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>int read(char buffer[ ])</td>
<td>Attempts to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read. −1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>int read(CharBuffer buffer)</td>
<td>Attempts to read characters into buffer and returns the actual number of characters that were successfully read. −1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>abstract int read(char buffer[ ], int offset, int numChars)</td>
<td>Attempts to read up to numChars characters into buffer starting at buffer[offset], returning the number of characters successfully read. −1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>boolean ready( )</td>
<td>Returns true if the next input request will not wait. Otherwise, it returns false.</td>
</tr>
<tr>
<td>void reset( )</td>
<td>Resets the input pointer to the previously set mark.</td>
</tr>
<tr>
<td>long skip(long numChars)</td>
<td>Skips over numChars characters of input, returning the number of characters actually skipped.</td>
</tr>
</tbody>
</table>

Table 19-3 The Methods Defined by Reader
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writer append(char *ch)</td>
<td>Appends <em>ch</em> to the end of the invoking output stream. Returns a reference to the invoking stream.</td>
</tr>
<tr>
<td>Writer append(CharSequence *chars)</td>
<td>Appends <em>chars</em> to the end of the invoking output stream. Returns a reference to the invoking stream.</td>
</tr>
<tr>
<td>Writer append(CharSequence *chars, int begin, int end)</td>
<td>Appends the subrange of <em>chars</em> specified by <em>begin</em> and <em>end</em>–1 to the end of the invoking output stream. Returns a reference to the invoking stream.</td>
</tr>
<tr>
<td>abstract void close()</td>
<td>Closes the output stream. Further write attempts will generate an IOException.</td>
</tr>
<tr>
<td>abstract void flush()</td>
<td>Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.</td>
</tr>
<tr>
<td>void write(int *ch)</td>
<td>Writes a single character to the invoking output stream. Note that the parameter is an int, which allows you to call write with an expression without having to cast it back to char. However, only the low-order 16 bits are written.</td>
</tr>
<tr>
<td>void write(char *buffer[])</td>
<td>Writes a complete array of characters to the invoking output stream.</td>
</tr>
<tr>
<td>abstract void write(char *buffer[], int offset, int numChars)</td>
<td>Writes a subrange of <em>numChars</em> characters from the array <em>buffer</em>, beginning at <em>buffer</em>[offset] to the invoking output stream.</td>
</tr>
<tr>
<td>void write(String *str)</td>
<td>Writes <em>str</em> to the invoking output stream.</td>
</tr>
<tr>
<td>void write(String *str, int offset, int numChars)</td>
<td>Writes a subrange of <em>numChars</em> characters from the string <em>str</em>, beginning at the specified <em>offset</em>.</td>
</tr>
</tbody>
</table>

Table 19-4 The Methods Defined by Writer

FileReader

The FileReader class creates a Reader that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

```
FileReader(String filePath)
FileReader(File fileObj)
```

Either can throw a FileNotFoundException. Here, *filePath* is the full path name of a file, and *fileObj* is a File object that describes the file.
The following example shows how to read lines from a file and display them on the standard output device. It reads its own source file, which must be in the current directory.

```java
// Demonstrate FileReader.
// This program uses try-with-resources. It requires JDK 7 or later.
import java.io.*;

class FileReaderDemo {
    public static void main(String args[]) {
        try (FileReader fr = new FileReader("FileReaderDemo.java") ) {
            int c;

            // Read and display the file.
            while((c = fr.read()) != -1) System.out.print((char) c);
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```

FileWriter

FileWriter creates a Writer that you can use to write to a file. Four of its most commonly used constructors are shown here:

- `FileWriter(String filePath)`
- `FileWriter(String filePath, boolean append)`
- `FileWriter(File fileObj)`
- `FileWriter(File fileObj, boolean append)`

They can all throw an IOException. Here, `filePath` is the full path name of a file, and `fileObj` is a File object that
describes the file. If append is true, then output is appended to the end of the file.

Creation of a FileWriter is not dependent on the file already existing. FileWriter will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an IOException will be thrown.

The following example is a character stream version of an example shown earlier when FileOutputStream was discussed. This version creates a sample buffer of characters by first making a String and then using the getChars( ) method to extract the character array equivalent. It then creates three files. The first, file1.txt, will contain every other character from the sample. The second, file2.txt, will contain the entire set of characters. Finally, the third, file3.txt, will contain only the last quarter.
CharArrayReader

CharArrayReader is an implementation of an input stream that uses a character array as the source. This class has two constructors, each of which requires a character array to provide the data source:

**CharArrayReader(char array [])**

**CharArrayReader(char array [], int start, int**
Here, array is the input source. The second constructor creates a Reader from a subset of your character array that begins with the character at the index specified by start and is numChars long.

The close() method implemented by CharArrayReader does not throw any exceptions. This is because it cannot fail.

The following example uses a pair of CharArrayReaders:
The input1 object is constructed using the entire lowercase alphabet, whereas input2 contains only the first five letters. Here is the output:

input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is:
abcde
CharArrayWriter

CharArrayWriter is an implementation of an output stream that uses an array as the destination. CharArrayWriter has two constructors, shown here:

```
CharArrayWriter()
CharArrayWriter(int numChars)
```

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by numChars. The buffer is held in the buf field of CharArrayWriter. The buffer size will be increased automatically, if needed. The number of characters held by the buffer is contained in the count field of CharArrayWriter. Both buf and count are protected fields.

The close( ) method has no effect on a CharArrayWriter.

The following example demonstrates CharArrayWriter by reworking the sample program shown earlier for ByteArrayOutputStream. It produces the same output as the previous version.
// Demonstrate CharArrayWriter.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class CharArrayWriterDemo {
    public static void main(String args[]) throws IOException {
        CharArrayWriter f = new CharArrayWriter();
        String s = "This should end up in the array";
        char buf[] = new char[s.length()];

        s.getChars(0, s.length(), buf, 0);

        try {
            f.write(buf);
        } catch (IOException e) {
            System.out.println("Error Writing to Buffer");
            return;
        }
    }
}
BufferedReader

BufferedReader improves performance by buffering input. It has two constructors:

BufferedReader(Reader inputStream)
BufferedReader(Reader inputStream, int bufSize)

The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in bufSize.

Closing a BufferedReader also causes the underlying stream specified by inputStream to be closed.
As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer. To support this, BufferedReader implements the mark( ) and reset( ) methods, and BufferedReader.markSupported( ) returns true.

The following example reworks the BufferedReader example, shown earlier, so that it uses a BufferedReader character stream rather than a buffered byte stream. As before, it uses the mark( ) and reset( ) methods to parse a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the reset( ) happens and where it does not. Output is the same as that shown earlier.
import java.io.*;

class BufferedReaderDemo {
    public static void main(String args[]) throws IOException {
        String s = "This is a © copyright symbol " +
            "but this is © not.\n";
        char buf[] = new char[s.length()];
        s.getChars(0, s.length(), buf, 0);

        CharArrayReader in = new CharArrayReader(buf);
        int c;
        boolean marked = false;

        try (BufferedReader f = new BufferedReader(in)) {
        }
BufferedWriter

A BufferedWriter is a Writer that buffers output. Using a BufferedWriter can improve performance by reducing the number of times data is actually physically written to the output device.
A BufferedWriter has these two constructors:

```java
BufferedWriter(Writer outputStream)
BufferedWriter(Writer outputStream, int bufSize)
```

The first form creates a buffered stream using a buffer with a default size. In the second, the size of the buffer is passed in `bufSize`.

PushbackReader

The PushbackReader class allows one or more characters to be returned to the input stream. This allows you to look ahead in the input stream. Here are its two constructors:

```java
PushbackReader(Reader inputStream)
PushbackReader(Reader inputStream, int bufSize)
```

The first form creates a buffered stream that allows one character to be pushed back. In the second, the size of the pushback buffer is passed in `bufSize`.

Closing a PushbackReader also closes the underlying stream specified by `inputStream`.

PushbackReader provides `unread( )`, which returns one or more characters to the invoking input stream. It has the three forms shown here:

```java
void unread(int ch) throws IOException
void unread(char buffer [ ]) throws IOException
void unread(char buffer [ ], int offset, int numChars)
```
The first form pushes back the character passed in `ch`. This will be the next character returned by a subsequent call to `read()`.
The second form returns the characters in `buffer`. The third form pushes back `numChars` characters beginning at `offset` from `buffer`. An `IOException` will be thrown if there is an attempt to return a character when the pushback buffer is full.

The following program reworks the earlier `PushbackInputStream` example by replacing `PushbackInputStream` with `PushbackReader`. As before, it shows how a programming language parser can use a pushback stream to deal with the difference between the `=` operator for comparison and the `=` operator for assignment.
import java.io.*;

class PushbackReaderDemo {
    public static void main(String args[]) {
        String s = "if (a == 4) a = 0;\n";
        char buf[] = new char[s.length()];
        s.getChars(0, s.length(), buf, 0);
        CharArrayReader in = new CharArrayReader(buf);

        int c;
        try (PushbackReader f = new PushbackReader(in)) {
            while ((c = f.read()) != -1) {
                switch(c) {
                case '=':
                    if ((c = f.read()) == '=')
                        System.out.print("\=\=\n");
                    else {
                        System.out.print("<=\n");
                        f.unread(c);
                    }
                break;
                default:
                    System.out.print((char) c);
                break;
                }
            }
        } catch (IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}

PrintWriter
PrintWriter is essentially a character-oriented version of PrintStream. It implements the Appendable, AutoCloseable, Closeable, and Flushable interfaces. PrintWriter has several constructors. The following have been supplied by PrintWriter from the start:
Here, outputStream specifies an open OutputStream that will receive output. The flushOnNewline parameter controls whether the output buffer is automatically flushed every time println(), printf(), or format() is called. If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic. Constructors that do not specify the flushOnNewline parameter do not automatically flush.

The next set of constructors gives you an easy way to construct a PrintWriter that writes its output to a file.

`PrintWriter(File outputFile)` throws FileNotFoundException

`PrintWriter(File outputFile, String charSet)`

throws FileNotFoundException, UnsupportedEncodingException

`PrintWriter(String outputFileName)` throws FileNotFoundException

`PrintWriter(String outputFileName, String charSet)`

throws FileNotFoundException, UnsupportedEncodingException
These allow a PrintWriter to be created from a File object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the PrintWriter object directs all output to the specified file. You can specify a character encoding by passing its name in charSet.

PrintWriter supports the print( ) and println( ) methods for all types, including Object. If an argument is not a primitive type, the PrintWriter methods will call the object’s toString( ) method and then output the result.

PrintWriter also supports the printf( ) method. It works the same way it does in the PrintStream class described earlier: It allows you to specify the precise format of the data. Here is how printf( ) is declared in PrintWriter:

```java
PrintWriter printf(String fmtString, Object ...args)
PrintWriter printf(Locale loc, String fmtString, Object ... args)
```

The first version writes args to standard output in the format specified by fmtString, using the default locale. The second lets you specify a locale. Both return the invoking PrintWriter.

The format( ) method is also supported. It has these general forms:
PrintWriter format(String fmtString, Object ... args)
PrintWriter format(Locale loc, String fmtString, Object ... args)

It works exactly like printf( ).

The Console Class
Recently (in JDK 6), the Console class was added to java.io. It is used to read from and write to the console, if one exists. It implements the Flushable interface. Console is primarily a convenience class because most of its functionality is available through System.in and System.out. However, its use can simplify some types of console interactions, especially when reading strings from the console.

Console supplies no constructors. Instead, a Console object is obtained by calling System.console( ), which is shown here:

    static Console console( )

If a console is available, then a reference to it is returned. Otherwise, null is returned. A console will not be available in all cases. Thus, if null is returned, no console I/O is possible.

Console defines the methods shown in Table 19-5. Notice that the input methods, such as readLine( ), throw IOException if an input error occurs. IOException is a subclass of Error. It indicates an I/O failure that is beyond the control of your program. Thus, you will not normally
catch an IOError. Frankly, if an IOError is thrown while accessing the console, it usually means there has been a catastrophic system failure.

Also notice the readPassword() methods. These methods let your application read a password without echoing what is typed. When reading passwords, you should “zero-out” both the array that holds the string entered by the user and the array that holds the password that the string is tested against. This reduces the chance that a malicious program will be able to obtain a password by scanning memory.
Table 19-5 The Methods Defined by Console

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void flush()</td>
<td>Causes buffered output to be written physically to the console.</td>
</tr>
<tr>
<td>Console format(String fmtString, Object...args)</td>
<td>Writes args to the console using the format specified by fmtString.</td>
</tr>
<tr>
<td>Console printf(String fmtString, Object...args)</td>
<td>Writes args to the console using the format specified by fmtString.</td>
</tr>
<tr>
<td>Reader reader()</td>
<td>Returns a reference to a Reader connected to the console.</td>
</tr>
<tr>
<td>String readLine()</td>
<td>Reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, null is returned. An IOException is thrown on failure.</td>
</tr>
<tr>
<td>String readLine(String fmtString, Object...args)</td>
<td>Displays a prompting string formatted as specified by fmtString and args, and then reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, null is returned. An IOException is thrown on failure.</td>
</tr>
<tr>
<td>char[] readPassword()</td>
<td>Reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, null is returned. An IOException is thrown on failure.</td>
</tr>
<tr>
<td>char[] readPassword(String fmtString, Object... args)</td>
<td>Displays a prompting string formatted as specified by fmtString and args, and then reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, null is returned. An IOException is thrown on failure.</td>
</tr>
<tr>
<td>PrintWriter writer()</td>
<td>Returns a reference to a Writer connected to the console.</td>
</tr>
</tbody>
</table>

Here is an example that demonstrates the Console class:

```
// Demonstrate Console.
import java.io.*;

class ConsoleDemo {
    public static void main(String args[]) {
        String str;
        Console con;
```
Here is sample output:

Enter a string: This is a test.
Here is your string: This is a test.

Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it. (More information about RMI appears in Chapter 28.)

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be
circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

An overview of the interfaces and classes that support serialization follows.

Serializable

Only an object that implements the Serializable interface can be saved and restored by the serialization facilities. The Serializable interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

Variables that are declared as transient are not saved by the serialization facilities. Also, static variables are not saved.

Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and
restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The Externalizable interface is designed for these situations.

The Externalizable interface defines these two methods:

```java
void readExternal(ObjectInput inStream)
  throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
  throws IOException
```

In these methods, `inStream` is the byte stream from which the object is to be read, and `outStream` is the byte stream to which the object is to be written.

ObjectOutput

The ObjectOutput interface extends the DataOutput and AutoCloseable interfaces and supports object serialization. It defines the methods shown in Table 19-6. Note especially the writeObject( ) method. This is called to serialize an object. All of these methods will throw an IOException on error conditions.
ObjectOutput

The ObjectOutput class extends the OutputStream class and implements the ObjectOutput interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

The argument `outStream` is the output stream to which serialized objects will be written. Closing an ObjectOutputStream automatically closes the underlying stream specified by `outStream`.

Several commonly used methods in this class are shown in Table 19-7. They will throw an IOException on error conditions. There is also an inner class to ObjectOutputStream called PutField. It facilitates the writing of persistent fields, and its use is beyond the scope of this book.

### Table 19-6 The Methods Defined by ObjectOutput

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close()</td>
<td>Closes the invoking stream. Further write attempts will generate an IOException.</td>
</tr>
<tr>
<td>void flush()</td>
<td>Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.</td>
</tr>
<tr>
<td>void write(byte buffer[], int offset, int numBytes)</td>
<td>Writes a subrange of <code>numBytes</code> bytes from the array <code>buffer</code>, beginning at <code>buffer[offset]</code>.</td>
</tr>
<tr>
<td>void write(int b)</td>
<td>Writes a single byte to the invoking stream. The byte written is the low-order byte of <code>b</code>.</td>
</tr>
<tr>
<td>void writeObject(Object obj)</td>
<td>Writes object <code>obj</code> to the invoking stream.</td>
</tr>
</tbody>
</table>

ObjectOutputStream

The ObjectOutputStream class extends the OutputStream class and implements the ObjectOutput interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

The argument `outStream` is the output stream to which serialized objects will be written. Closing an ObjectOutputStream automatically closes the underlying stream specified by `outStream`.

Several commonly used methods in this class are shown in Table 19-7. They will throw an IOException on error conditions. There is also an inner class to ObjectOutputStream called PutField. It facilitates the writing of persistent fields, and its use is beyond the scope of this book.
### Table 19-7 A Sampling of Commonly Used Methods Defined by ObjectOutputStream

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void close()</td>
<td>Closes the invoking stream. Further write attempts will generate an IOException. The underlying stream is also closed.</td>
</tr>
<tr>
<td>void flush()</td>
<td>Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.</td>
</tr>
<tr>
<td>void write(byte buffer[], int offset, int numBytes)</td>
<td>Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].</td>
</tr>
<tr>
<td>void write(int b)</td>
<td>Writes a single byte to the invoking stream. The byte written is the low-order byte of b.</td>
</tr>
<tr>
<td>void writeBoolean(boolean b)</td>
<td>Writes a boolean to the invoking stream.</td>
</tr>
<tr>
<td>void writeByte(int b)</td>
<td>Writes a byte to the invoking stream. The byte written is the low-order byte of b.</td>
</tr>
<tr>
<td>void writeBytes(String str)</td>
<td>Writes the bytes representing str to the invoking stream.</td>
</tr>
<tr>
<td>void writeChar(int c)</td>
<td>Writes a char to the invoking stream.</td>
</tr>
<tr>
<td>void writeChars(String str)</td>
<td>Writes the characters in str to the invoking stream.</td>
</tr>
<tr>
<td>void writeDouble(double d)</td>
<td>Writes a double to the invoking stream.</td>
</tr>
<tr>
<td>void writeFloat(float f)</td>
<td>Writes a float to the invoking stream.</td>
</tr>
<tr>
<td>void writeInt(int i)</td>
<td>Writes an int to the invoking stream.</td>
</tr>
<tr>
<td>void writeLong(long l)</td>
<td>Writes a long to the invoking stream.</td>
</tr>
<tr>
<td>final void writeObject(Object obj)</td>
<td>Writes obj to the invoking stream.</td>
</tr>
<tr>
<td>void writeShort(int i)</td>
<td>Writes a short to the invoking stream.</td>
</tr>
</tbody>
</table>

ObjectInput

The ObjectInput interface extends the DataInput and AutoCloseable interfaces and defines the methods shown in Table 19-8. It supports object serialization. Note especially the readObject( ) method. This is called to deserialize an object. All of these methods will throw an IOException on error conditions. The readObject( ) method can also throw ClassNotFoundException.
The `ObjectInputStream` class extends the `InputStream` class and implements the `ObjectInput` interface. `ObjectInputStream` is responsible for reading objects from a stream. One constructor of this class is shown here:

```java
ObjectInputStream(InputStream inStream) throws IOException
```

The argument `inStream` is the input stream from which serialized objects should be read. Closing an `ObjectInputStream` automatically closes the underlying stream specified by `inStream`.

Several commonly used methods in this class are shown in `Table 19-9`. They will throw an `IOException` on error conditions. The `readObject()` method can also throw `ClassNotFoundException`. There is also an inner class to `ObjectInputStream` called `GetField`. It facilitates the reading of persistent fields, and its use is beyond the scope of this book.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int available( )</td>
<td>Returns the number of bytes that are now available in the input buffer.</td>
</tr>
<tr>
<td>void close( )</td>
<td>Closes the invoking stream. Further read attempts will generate an IOException.</td>
</tr>
<tr>
<td>int read( )</td>
<td>Returns an integer representation of the next available byte of input. −1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>int read(byte buffer[ ])</td>
<td>Attempts to read up to buffer.length bytes into buffer, returning the number of bytes that were successfully read. −1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>int read(byte buffer[ ], int offset, int numBytes)</td>
<td>Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes that were successfully read. −1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>Object readObject( )</td>
<td>Reads an object from the invoking stream.</td>
</tr>
<tr>
<td>long skip(long numBytes)</td>
<td>Ignores (that is, skips) numBytes bytes in the invoking stream, returning the number of bytes actually ignored.</td>
</tr>
</tbody>
</table>

Table 19-8 The Methods Defined by ObjectInput
Table 19-9 Commonly Used Methods Defined by ObjectInputStream

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int available( )</td>
<td>Returns the number of bytes that are now available in the input buffer.</td>
</tr>
<tr>
<td>void close( )</td>
<td>Closes the invoking stream. Further read attempts will generate an IOException. The underlying stream is also closed.</td>
</tr>
<tr>
<td>int read( )</td>
<td>Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>int read(byte buffer[ ], int offset, int numBytes)</td>
<td>Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. -1 is returned when the end of the file is encountered.</td>
</tr>
<tr>
<td>Boolean readBoolean( )</td>
<td>Reads and returns a boolean from the invoking stream.</td>
</tr>
<tr>
<td>byte readByte( )</td>
<td>Reads and returns a byte from the invoking stream.</td>
</tr>
<tr>
<td>char readChar( )</td>
<td>Reads and returns a char from the invoking stream.</td>
</tr>
<tr>
<td>double readDouble( )</td>
<td>Reads and returns a double from the invoking stream.</td>
</tr>
<tr>
<td>float readFloat( )</td>
<td>Reads and returns a float from the invoking stream.</td>
</tr>
<tr>
<td>void readFully(byte buffer[ ])</td>
<td>Reads buffer.length bytes into buffer. Returns only when all bytes have been read.</td>
</tr>
<tr>
<td>void readFully(byte buffer[ ], int offset, int numBytes)</td>
<td>Reads numBytes bytes into buffer starting at buffer[offset]. Returns only when numBytes have been read.</td>
</tr>
<tr>
<td>int readInt( )</td>
<td>Reads and returns an int from the invoking stream.</td>
</tr>
<tr>
<td>long readLong( )</td>
<td>Reads and returns a long from the invoking stream.</td>
</tr>
<tr>
<td>final Object readObject( )</td>
<td>Reads and returns an object from the invoking stream.</td>
</tr>
<tr>
<td>short readShort( )</td>
<td>Reads and returns a short from the invoking stream.</td>
</tr>
<tr>
<td>int readUnsignedByte( )</td>
<td>Reads and returns an unsigned byte from the invoking stream.</td>
</tr>
<tr>
<td>int readUnsignedShort( )</td>
<td>Reads and returns an unsigned short from the invoking stream.</td>
</tr>
</tbody>
</table>

A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class MyClass. This object has three instance variables that are of types String, int, and double. This is the information we want to save and restore.

A FileOutputStream is created that refers to a file
named “serial”, and an ObjectOutputStream is created for that file stream. The writeObject( ) method of ObjectOutputStream is then used to serialize our object. The object output stream is flushed and closed.

A FileInputStream is then created that refers to the file named “serial”, and an ObjectInputStream is created for that file stream. The readObject( ) method of ObjectInputStream is then used to deserialize our object. The object input stream is then closed.

Note that MyClass is defined to implement the Serializable interface. If this is not done, a NotSerializableException is thrown. Try experimenting with this program by declaring some of the MyClass instance variables to be transient. That data is then not saved during serialization.
import java.io.*;

public class SerializationDemo {
    public static void main(String args[]) {

        // Object serialization

        try (ObjectOutputStream objOStrm =
             new ObjectOutputStream(new FileOutputStream("serial")))
        {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            System.out.println("object1: " + object1);

            objOStrm.writeObject(object1);
        }
        catch(IOException e) {
            System.out.println("Exception during serialization: " + e);
        }

        // Object deserialization

        try (ObjectInputStream objIStrm =
             new ObjectInputStream(new FileInputStream("serial")))
        {
            MyClass object2 = (MyClass)objIStrm.readObject();
            System.out.println("object2: " + object2);
        }
        catch(Exception e) {
            System.out.println("Exception during deserialization: " + e);
        }
    }
}
This program demonstrates that the instance variables of `object1` and `object2` are identical. The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Stream Benefits

The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task. The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements. Java programs written to adhere to the abstract, high-level `InputStream`, `OutputStream`, `Reader`, and `Writer` classes will function properly in the future even when new and improved concrete stream classes are invented. As you will see in Chapter 21, this model works very well when we switch from a file system–based set of streams to the network and socket streams. Finally, serialization of objects plays an important role in many types of Java programs.
Java’s serialization I/O classes provide a portable solution to this sometimes tricky task.
CHAPTER 20
Exploring NIO

Beginning with version 1.4, Java has provided a second I/O system called NIO (which is short for New I/O). It supports a buffer-oriented, channel-based approach to I/O operations. With the release of JDK 7, the NIO system has been greatly expanded and now provides enhanced support for file-handling and file system features. In fact, so significant are the changes that the term NIO.2 is often used. Because of the capabilities provided by the new NIO file classes, NIO is expected to become an increasingly important part of file handling going forward. This chapter explores several of the key features of the NIO system, including its new file-handling capabilities.

The NIO Classes

The NIO classes are contained in the packages shown here:

<table>
<thead>
<tr>
<th>Package</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.nio</td>
<td>Top-level package for the NIO system. Encapsulates various types of buffers that contain data operated upon by the NIO system.</td>
</tr>
<tr>
<td>java.nio.channels</td>
<td>Supports channels, which are essentially open I/O connections.</td>
</tr>
<tr>
<td>java.nio.channels.spi</td>
<td>Supports service providers for channels.</td>
</tr>
<tr>
<td>java.nio.charset</td>
<td>Encapsulates character sets. Also supports encoders and decoders that convert characters to bytes and bytes to characters, respectively.</td>
</tr>
<tr>
<td>java.nio.charset.spi</td>
<td>Supports service providers for character sets.</td>
</tr>
<tr>
<td>java.nio.file</td>
<td>Provides support for files. (Added by JDK 7.)</td>
</tr>
<tr>
<td>java.nio.file.attribute</td>
<td>Provides support for file attributes. (Added by JDK 7.)</td>
</tr>
<tr>
<td>java.nio.file.spi</td>
<td>Supports service providers for file systems. (Added by JDK 7.)</td>
</tr>
</tbody>
</table>

Before we begin, it is important to emphasize that the NIO subsystem does not replace the stream-based I/O
classes found in java.io, which are discussed in Chapter 19, and good working knowledge of the stream-based I/O in java.io is helpful to understanding NIO.

NOTE
This chapter assumes that you have read the overview of I/O given in Chapter 13 and the discussion of stream-based I/O supplied in Chapter 19.

NIO Fundamentals
The NIO system is built on two foundational items: buffers and channels. A buffer holds data. A channel represents an open connection to an I/O device, such as a file or a socket. In general, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold data. You then operate on the buffer, inputting or outputting data as needed. The following sections examine buffers and channels in more detail.

Buffers
Buffers are defined in the java.nio package. All buffers are subclasses of the Buffer class, which defines the core functionality common to all buffers: current position, limit, and capacity. The current position is the index within the buffer at which the next read or write operation will take place. The current position is advanced by most read or write operations. The limit is the index value one past the last valid location in the buffer. The capacity is the number of elements that the
buffer can hold. Often the limit equals the capacity of the buffer. Buffer also supports mark and reset. Buffer defines several methods, which are shown in Table 20-1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract Object array( )</td>
<td>If the invoking buffer is backed by an array, returns a reference to the array. Otherwise, an UnsupportedOperationException is thrown. If the array is read-only, a ReadOnlyBufferException is thrown.</td>
</tr>
<tr>
<td>abstract int arrayOffset( )</td>
<td>If the invoking buffer is backed by an array, returns the index of the first element. Otherwise, an UnsupportedOperationException is thrown. If the array is read-only, a ReadOnlyBufferException is thrown.</td>
</tr>
<tr>
<td>final int capacity( )</td>
<td>Returns the number of elements that the invoking buffer is capable of holding.</td>
</tr>
<tr>
<td>final Buffer clear( )</td>
<td>Clears the invoking buffer and returns a reference to the buffer.</td>
</tr>
<tr>
<td>final Buffer flip( )</td>
<td>Sets the invoking buffer’s limit to the current position and resets the current position to 0. Returns a reference to the buffer.</td>
</tr>
<tr>
<td>abstract boolean hasArray( )</td>
<td>Returns true if the invoking buffer is backed by a read/write array and false otherwise.</td>
</tr>
<tr>
<td>final boolean hasRemaining( )</td>
<td>Returns true if there are elements remaining in the invoking buffer. Returns false otherwise.</td>
</tr>
<tr>
<td>abstract boolean isDirect( )</td>
<td>Returns true if the invoking buffer is direct, which means I/O operations act directly upon it. Returns false otherwise.</td>
</tr>
<tr>
<td>abstract boolean isReadOnly( )</td>
<td>Returns true if the invoking buffer is read-only. Returns false otherwise.</td>
</tr>
<tr>
<td>final int limit( )</td>
<td>Returns the invoking buffer’s limit.</td>
</tr>
<tr>
<td>final Buffer limit(int n)</td>
<td>Sets the invoking buffer’s limit to n. Returns a reference to the buffer.</td>
</tr>
<tr>
<td>final Buffer mark( )</td>
<td>Sets the mark and returns a reference to the invoking buffer.</td>
</tr>
<tr>
<td>final int position( )</td>
<td>Returns the current position.</td>
</tr>
<tr>
<td>final Buffer position(int n)</td>
<td>Sets the invoking buffer’s current position to n. Returns a reference to the buffer.</td>
</tr>
<tr>
<td>int remaining( )</td>
<td>Returns the number of elements available before the limit is reached. In other words, it returns the limit minus the current position.</td>
</tr>
<tr>
<td>final Buffer reset( )</td>
<td>Resets the current position of the invoking buffer to the previously set mark. Returns a reference to the buffer.</td>
</tr>
<tr>
<td>final Buffer rewind( )</td>
<td>Sets the position of the invoking buffer to 0. Returns a reference to the buffer.</td>
</tr>
</tbody>
</table>

Table 20-1 The Methods Defined by Buffer

From Buffer, the following specific buffer classes are
derived, which hold the type of data that their names imply:

<table>
<thead>
<tr>
<th>ByteBuffer</th>
<th>CharBuffer</th>
<th>DoubleBuffer</th>
<th>FloatBuffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntBuffer</td>
<td>LongBuffer</td>
<td>MappedByteBuffer</td>
<td>ShortBuffer</td>
</tr>
</tbody>
</table>

MappedByteBuffer is a subclass of ByteBuffer and is used to map a file to a buffer.

All buffers provide various `get()` and `put()` methods, which allow you to get data from a buffer or put data into a buffer. (Of course, if a buffer is read-only, then `put()` operations are not available.) Table 20-2 shows the `get()` and `put()` methods defined by ByteBuffer. The other buffer classes have similar methods. All buffer classes also support methods that perform various buffer operations. For example, you can allocate a buffer manually using `allocate()`. You can wrap an array inside a buffer using `wrap()`. You can create a subsequence of a buffer using `slice()`.

Channels

Channels are defined in java.nio.channels. A channel represents an open connection to an I/O source or destination. Channels implement the Channel interface. It extends Closeable, and, beginning with JDK 7, it extends AutoCloseable. By implementing AutoCloseable, channels can be managed by JDK 7’s new `try-with-resources` statement. When used in a `try-with-resources` block, a channel is closed automatically when it is no longer needed. (See Chapter 13 for a discussion of `try-with-resources`.)
Table 20-2 The get( ) and put( ) Methods Defined for ByteBuffer

One way to obtain a channel is by calling getChannel( ) on an object that supports channels. For example, getChannel( ) is supported by the following I/O classes:

<table>
<thead>
<tr>
<th>DatagramSocket</th>
<th>FileInputStream</th>
<th>FileOutputStream</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomAccessFile</td>
<td>ServerSocket</td>
<td>Socket</td>
</tr>
</tbody>
</table>

The specific type of channel returned depends upon the type of object getChannel( ) is called on. For example, when called on a FileInputStream,
FileOutputStream, or RandomAccessFile, getChannel( ) returns a channel of type FileChannel. When called on a Socket, getChannel( ) returns a SocketChannel.

Another way to obtain a channel is to use one of the static methods defined by the Files class, which was added by JDK 7. For example, using Files, you can obtain a byte channel by calling newByteChannel( ). It returns a SeekableByteChannel, which is an interface implemented by FileChannel. (The Files class is examined in detail later in this chapter.)

Channels such as FileChannel and SocketChannel support various read( ) and write( ) methods that enable you to perform I/O operations through the channel. For example, here are a few of the read( ) and write( ) methods defined for FileChannel.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract int read(ByteBuffer bb) throws IOException</td>
<td>Reads bytes from the invoking channel into bb until the buffer is full or there is no more input. Returns the number of bytes actually read.</td>
</tr>
<tr>
<td>abstract int read(ByteBuffer bb, long start) throws IOException</td>
<td>Beginning at the file location specified by start, reads bytes from the invoking channel into bb until the buffer is full or there is no more input. The current position is unchanged. Returns the number of bytes actually read or -1 if start is beyond the end of the file.</td>
</tr>
<tr>
<td>abstract int write(ByteBuffer bb) throws IOException</td>
<td>Writes the contents of bb to the invoking channel, starting at the current position. Returns the number of bytes written.</td>
</tr>
<tr>
<td>abstract int write(ByteBuffer bb, long start) throws IOException</td>
<td>Beginning at the file location specified by start, writes the contents of bb to the invoking channel. The current position is unchanged. Returns the number of bytes written.</td>
</tr>
</tbody>
</table>

All channels support additional methods that give you access to and control over the channel. For example, FileChannel supports methods to get or set the current position, transfer information between file channels,
obtain the current size of the channel, and lock the channel, among others. Beginning with JDK 7, FileChannel provides a static method called `open()`, which opens a file and returns a channel to it. This provides another way to obtain a channel. FileChannel also provides the `map()` method, which lets you map a file to a buffer.

Charsets and Selectors

Two other entities used by NIO are charsets and selectors. A charset defines the way that bytes are mapped to characters. You can encode a sequence of characters into bytes using an encoder. You can decode a sequence of bytes into characters using a decoder.Charsets, encoders, and decoders are supported by classes defined in the `java.nio.charset` package. Because default encoders and decoders are provided, you will not often need to work explicitly with charsets.

A selector supports key-based, non-blocking, multiplexed I/O. In other words, selectors enable you to perform I/O through multiple channels. Selectors are supported by classes defined in the `java.nio.channels` package. Selectors are most applicable to socket-backed channels.

We will not use charsets or selectors in this chapter, but you might find them useful in your own applications.

Enhancements Added to NIO by JDK 7

Beginning with JDK 7, the NIO system was substantially
expanded and enhanced. In addition to support for the try-with-resources statement (which provides automatic resource management), the improvements include three new packages (java.nio.file, java.nio.file.attribute, and java.nio.file.spi); several new classes, interfaces, and methods; and direct support for stream-based I/O. The additions have greatly expanded the ways in which NIO can be used, especially with files. Several of the key additions are described in the following sections.

The Path Interface

Perhaps the single most important addition to the NIO system is the Path interface because it encapsulates a path to a file. As you will see, Path is the glue that binds together many of the new NIO.2 file-based features. It describes a file’s location within the directory structure. Path is packaged in java.nio.file, and it inherits the following interfaces: Watchable, Iterable<Path>, and Comparable<Path>. Watchable describes an object that can be monitored for changes. It was also added by JDK 7. The Iterable and Comparable interfaces were described earlier in this book.

Path declares a number of methods that operate on the path. A sampling is shown in Table 20-3. Pay special attention to the getName() method. It is used to obtain an element in a path. It works using an index. At index zero, is the part of the path nearest the root, which is the leftmost element in a path. Subsequent indexes specify elements to the right of the root. The number of elements in a path can be obtained by calling
getNameCount( ). If you want to obtain a string representation of the entire path, simply call toString( ). Notice that you can resolve a relative path into an absolute path by using the resolve( ) method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean endsWith(String path)</td>
<td>Returns true if the invoking Path ends with the path specified by path. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean endsWith(Path path)</td>
<td>Returns true if the invoking Path ends with the path specified by path. Otherwise, returns false.</td>
</tr>
<tr>
<td>Path getName( )</td>
<td>Returns the filename associated with the invoking Path.</td>
</tr>
<tr>
<td>Path getName(int idx)</td>
<td>Returns a Path object that contains the name of the path element specified by idx within the invoking object. The leftmost element is at index 0. This is the element nearest the root. The rightmost element is at getNameCount( ) – 1.</td>
</tr>
<tr>
<td>int getNameCount( )</td>
<td>Returns the number of elements beyond the root directory in the invoking Path.</td>
</tr>
<tr>
<td>Path getParent( )</td>
<td>Returns a Path that contains the entire path except for the name of the file specified by the invoking Path.</td>
</tr>
<tr>
<td>Path getRoot( )</td>
<td>Returns the root of the invoking Path.</td>
</tr>
<tr>
<td>boolean isAbsolute( )</td>
<td>Returns true if the invoking Path is absolute. Otherwise, returns false.</td>
</tr>
<tr>
<td>Path resolve(Path path)</td>
<td>If path is absolute, path is returned. Otherwise, if path does not contain a root, path is prefixed by the root specified by the invoking Path and the result is returned. If path is empty, the invoking Path is returned. Otherwise, the behavior is unspecified.</td>
</tr>
<tr>
<td>Path resolve(String path)</td>
<td>If path is absolute, path is returned. Otherwise, if path does not contain a root, path is prefixed by the root specified by the invoking Path and the result is returned. If path is empty, the invoking Path is returned. Otherwise, the behavior is unspecified.</td>
</tr>
<tr>
<td>boolean startsWith(String path)</td>
<td>Returns true if the invoking Path starts with the path specified by path. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean startsWith(Path path)</td>
<td>Returns true if the invoking Path starts with the path specified by path. Otherwise, returns false.</td>
</tr>
<tr>
<td>Path toAbsolutePath( )</td>
<td>Returns the invoking Path as an absolute path.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns a string representation of the invoking Path.</td>
</tr>
</tbody>
</table>

Table 20-3 A Sampling of Methods Specified by Path

One other point: When updating legacy code that uses the File class defined by java.io, it is possible to convert
a File instance into a Path instance by calling toPath( ) on the File object. This method was added to File by JDK 7. Furthermore, it is possible to obtain a File instance by calling the toFile( ) method defined by Path.

The Files Class

Many of the actions that you perform on a file are provided by static methods within the Files class. The file to be acted upon is specified by its Path. Thus, the Files methods use a Path to specify the file that is being operated upon. Files contains a wide array of functionality. For example, it has methods that let you open or create a file that has the specified path. You can obtain information about a Path, such as whether it is executable, hidden, or read-only. Files also supplies methods that let you copy or move files. A sampling is shown in Table 20-4. In addition to IOException, several other exceptions are possible.

Notice that several of the methods in Table 20-4 take an argument of type OpenOption. This is an interface that describes how to open a file. It is implemented by the StandardOpenOption class, which defines an enumeration that has the values shown in Table 20-5.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static Path copy(Path src, Path dest,</td>
<td>Copies the file specified by src to the location specified by dest. The how parameter specifies how the copy will take place.</td>
</tr>
<tr>
<td>CopyOption ... how) throws IOException</td>
<td></td>
</tr>
<tr>
<td>static Path createDirectory(Path path,</td>
<td>Creates the directory whose path is specified by path. The directory attributes are specified by attrs.</td>
</tr>
<tr>
<td>FileAttribute&lt;?&gt; ... attrs) throws</td>
<td></td>
</tr>
<tr>
<td>IOException</td>
<td></td>
</tr>
<tr>
<td>static Path createFile(Path path,</td>
<td>Creates the file whose path is specified by path. The file attributes are specified by attrs.</td>
</tr>
<tr>
<td>FileAttribute&lt;?&gt; ... attrs) throws</td>
<td></td>
</tr>
<tr>
<td>IOException</td>
<td></td>
</tr>
<tr>
<td>static void delete(Path path) throws</td>
<td>Deletes the file whose path is specified by path.</td>
</tr>
<tr>
<td>IOException</td>
<td></td>
</tr>
<tr>
<td>static boolean exists(Path path,</td>
<td>Returns true if the file specified by path exists and false otherwise. If opts is not specified, then symbolic links are followed. To prevent</td>
</tr>
<tr>
<td>LinkOptions ... opts)</td>
<td>the following of symbolic links, pass NOFOLLOW_LINKS to opts.</td>
</tr>
<tr>
<td>static boolean isDirectory(Path path,</td>
<td>Returns true if path specifies a directory and false otherwise. If opts is not specified, then symbolic links are followed. To prevent the</td>
</tr>
<tr>
<td>LinkOptions ... opts)</td>
<td>following of symbolic links, pass NOFOLLOW_LINKS to opts.</td>
</tr>
<tr>
<td>static boolean isExecutable(Path path)</td>
<td>Returns true if the file specified by path is executable and false otherwise.</td>
</tr>
<tr>
<td>throws IOException</td>
<td></td>
</tr>
<tr>
<td>static boolean isHidden(Path path)</td>
<td>Returns true if the file specified by path is hidden and false otherwise.</td>
</tr>
<tr>
<td>throws IOException</td>
<td></td>
</tr>
<tr>
<td>static boolean isReadable(Path path)</td>
<td>Returns true if the file specified by path can be read from and false otherwise.</td>
</tr>
<tr>
<td>static boolean isRegularFile(Path path,</td>
<td>Returns true if path specifies a file and false otherwise. If opts is not specified, then symbolic links are followed. To prevent the following</td>
</tr>
<tr>
<td>LinkOptions ... opts)</td>
<td>of symbolic links, pass NOFOLLOW_LINKS to opts.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>static boolean isWritable(Path path)</td>
<td>Returns true if the file specified by path can be written to and false otherwise.</td>
</tr>
<tr>
<td>static Path move(Path src, Path dest, CopyOption ... how) throws IOException</td>
<td>Moves the file specified by src to the location specified by dest. The how parameter specifies how the move will take place.</td>
</tr>
<tr>
<td>static SeekableByteChannel newByteChannel(Path path, OpenOption ... how) throws IOException</td>
<td>Opens the file specified by path, as specified by how. Returns a SeekableByteChannel to the file. This is a byte channel whose current position can be changed. SeekableByteChannel is implemented by FileChannel.</td>
</tr>
<tr>
<td>static DirectoryStream&lt;Path&gt; newDirectoryStream(Path path) throws IOException</td>
<td>Opens the directory specified by path. Returns a DirectoryStream linked to the directory.</td>
</tr>
<tr>
<td>static InputStream newInputStream(Path path, OpenOption ... how) throws IOException</td>
<td>Opens the file specified by path, as specified by how. Returns an InputStream linked to the file.</td>
</tr>
<tr>
<td>static OutputStream newOutputStream(Path path, OpenOption ... how) throws IOException</td>
<td>Opens the file specified by the invoking object, as specified by how. Returns an OutputStream linked to the file.</td>
</tr>
<tr>
<td>static boolean notExists(Path path, LinkOption ... opts)</td>
<td>Returns true if the file specified by path does not exist and false otherwise. If opts is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass NOFOLLOW_LINKS to opts.</td>
</tr>
<tr>
<td>static &lt;A extends BasicFileAttributes&gt; A readAttributes(Path path, Class&lt;A&gt; attribType, LinkOption ... opts) throws IOException</td>
<td>Obtains the attributes associated with the file specified by path. The type of attributes to obtain is passed in attribType. If opts is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass NOFOLLOW_LINKS to opts.</td>
</tr>
<tr>
<td>static long size(Path path) throws IOException</td>
<td>Returns the size of the file specified by path.</td>
</tr>
</tbody>
</table>

Table 20-4 A Sampling of Methods Defined by Files
Table 20-5 The Standard Open Options

The Paths Class

Because Path is an interface, not a class, you can’t create an instance of Path directly through the use of a constructor. Instead, you obtain a Path by a calling a method that returns one. Frequently, you do this by using the get() method defined by the Paths class. There are two forms of get(). The one used in this chapter is shown here:

    static Path get(String pathname, String … parts)

It returns a Path that encapsulates the specified path. The path can be specified in two ways. First, if parts is not used, then the path must be specified in its entirety by pathname. Alternatively, you can pass the path in pieces, with the first part passed in pathname and the
subsequent elements specified by the parts varargs parameter. In either case, if the path specified is syntactically invalid, get() will throw an InvalidPathException.

The second form of get() creates a Path from a URI. It is shown here:

```
static Path get(URI uri)
```

The Path corresponding to uri is returned.

It is important to understand that creating a Path to a file does not open or create a file. It simply creates an object that encapsulates the file’s directory path.

The File Attribute Interfaces

Associated with a file is a set of attributes. These attributes include such things as the file’s time of creation, the time of its last modification, whether the file is a directory, and its size. NIO organizes file attributes into several different interfaces. Attributes are represented by a hierarchy of interfaces defined in java.nio.file.attribute. At the top is BasicFileAttributes. It encapsulates the set of attributes that are commonly found in a variety of file systems. The methods defined by BasicFileAttributes are shown in Table 20-6.
Table 20-6 The Methods Defined by BasicFileAttributes

From BasicFileAttributes two interfaces are derived: DosFileAttributes and PosixFileAttributes. DosFileAttributes describes those attributes related to the FAT file system as first defined by DOS. It defines the methods shown here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileTime creationTime( )</td>
<td>Returns the time at which the file was created. If creation time is not provided by the file system, then an implementation-dependent value is returned.</td>
</tr>
<tr>
<td>Object fileKey( )</td>
<td>Returns the file key. If not supported, <code>null</code> is returned.</td>
</tr>
<tr>
<td>boolean isDirectory( )</td>
<td>Returns <code>true</code> if the file represents a directory.</td>
</tr>
<tr>
<td>boolean isOther( )</td>
<td>Returns <code>true</code> if the file is not a file, symbolic link, or a directory.</td>
</tr>
<tr>
<td>boolean isRegularFile( )</td>
<td>Returns <code>true</code> if the file is a normal file, rather than a directory or symbolic link.</td>
</tr>
<tr>
<td>boolean isSymbolicLink( )</td>
<td>Returns <code>true</code> if the file is a symbolic link.</td>
</tr>
<tr>
<td>FileTime lastAccessTime( )</td>
<td>Returns the time at which the file was last accessed. If the time of last access is not provided by the file system, then an implementation-dependent value is returned.</td>
</tr>
<tr>
<td>FileTime lastModifiedTime( )</td>
<td>Returns the time at which the file was last modified. If the time of last modification is not provided by the file system, then an implementation-dependent value is returned.</td>
</tr>
<tr>
<td>long size( )</td>
<td>Returns the size of the file.</td>
</tr>
</tbody>
</table>

PosixFileAttributes encapsulates attributes defined by the POSIX standards. (POSIX stands for Portable Operating System Interface.) It defines the methods shown here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isArchive( )</td>
<td>Returns <code>true</code> if the file is flagged for archiving and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>boolean isHidden( )</td>
<td>Returns <code>true</code> if the file is hidden and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>boolean isReadOnly( )</td>
<td>Returns <code>true</code> if the file is read-only and <code>false</code> otherwise.</td>
</tr>
<tr>
<td>boolean isSystem( )</td>
<td>Returns <code>true</code> if the file is flagged as a system file and <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>

GroupPrincipal group( ) Returns the file’s group owner.
UserPrincipal owner( ) Returns the file’s owner.
Set<PosixFilePermission> permissions( ) Returns the file’s permissions.
There are various ways to access a file's attributes. First, you can obtain an object that encapsulates a file’s attributes by calling `readAttributes()`, which is a static method defined by `Files`. One of its forms is shown here:

```java
static <A extends BasicFileAttributes>
    A readAttributes(Path path, Class<A> attrType, LinkOption... opts)
    throws IOException
```

This method returns a reference to an object that specifies the attributes associated with the file passed in `path`. The specific type of attributes is specified as a `Class` object in the `attrType` parameter. For example, to obtain the basic file attributes, pass `BasicFileAttributes.class` to `attrType`. For DOS attributes, use `DosFileAttributes.class`, and for POSIX attributes, use `PosixFileAttributes.class`. Optional link options are passed via `opts`. If not specified, symbolic links are followed. The method returns a reference to requested attributes. If the requested attribute type is not available, `UnsupportedOperationException` is thrown. Using the object returned, you can access the file’s attributes.

A second way to gain access to a file’s attributes is to call `getFileAttributeView()` defined by `Files`. NIO defines several attribute view interfaces, including `AttributeView`, `BasicFileAttributeView`, `DosFileAttributeView`, and `PosixFileAttributeView`, among others. Although we won’t be using attribute views in this chapter, they are a feature that you may
In some cases, you won’t need to use the file attribute interfaces directly because the Files class offers static convenience methods that access several of the attributes. For example, Files includes methods such as isHidden( ) and isWritable( ).

It is important to understand that not all file systems support all possible attributes. For example, the DOS file attributes apply to the older FAT file system as first defined by DOS. The attributes that will apply to a wide variety of file systems are described by BasicFileAttributes. For this reason, these attributes are used in the examples in this chapter.

The FileSystem, FileSystems, and FileStore Classes

JDK 7 adds the ability to easily access the file system through the FileSystem and FileSystems classes packaged in java.nio.file. In fact, by using the newFileSystem( ) method defined by FileSystems, it is even possible to obtain a new file system. The FileStore class encapsulates the file storage system. Although these classes are not used directly in this chapter, you may find them helpful in your own applications.

Using the NIO System

This section illustrates how to apply the NIO system to a variety of tasks. Before beginning, it is important to emphasize that with the release of JDK 7, the NIO subsystem was greatly expanded. As a result, its uses have also been greatly expanded. As mentioned, the
enhanced version is sometimes referred to as NIO.2. Because the features added by NIO.2 are so substantial, they have changed the way that much NIO-based code is written and have increased the types of tasks to which NIO can be applied. Because of its importance, most of the remaining discussion and examples in this chapter utilize NIO.2 features and, therefore, require JDK 7 or later. However, at the end of the chapter is a brief description of pre-JDK 7 code. This will be of aid to those programmers working in pre-JDK 7 environments or maintaining older code.

REMEMBER

Most of the examples in this chapter require JDK 7 or later.

In the past, the primary purpose of NIO was channel-based I/O, and this is still a very important use. However, you can now use NIO for stream-based I/O and for performing file-system operations. As a result, the discussion of using NIO is divided into three parts:

• Using NIO for channel-based I/O
• Using NIO for stream-based I/O
• Using NIO for path and file system operations

Because the most common I/O device is the disk file, the rest of this chapter uses disk files in the examples. Because all file channel operations are byte-based, the type of buffers that we will be using are of type ByteBuffer.
Before you can open a file for access via the NIO system, you must obtain a Path that describes the file. One way to do this is to call the Paths.get( ) factory method, which was described earlier. The form of get( ) used in the examples is shown here:

```java
static Path get(String pathname, String ... parts)
```

Recall that the path can be specified in two ways. It can be passed in pieces, with the first part passed in pathname and the subsequent elements specified by the parts varargs parameter. Alternatively, the entire path can be specified in pathname and parts is not used. This is the approach used by the examples.

Use NIO for Channel-Based I/O

An important use of NIO is to access a file via a channel and buffers. The following sections demonstrate some techniques that use a channel to read from and write to a file.

Reading a File via a Channel

There are several ways to read data from a file using a channel. Perhaps the most common way is to manually allocate a buffer and then perform an explicit read operation that loads that buffer with data from the file. It is with this approach that we begin.

Before you can read from a file, you must open it. To do this, first create a Path that describes the file. Then use this Path to open the file. There are various ways to open the file depending on how it will be used. In this
example, the file will be opened for byte-based input via explicit input operations. Therefore, this example will open the file and establish a channel to it by calling Files.newByteChannel(). The newByteChannel() method has this general form:

```java
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
    throws IOException
```

It returns a SeekableByteChannel object, which encapsulates the channel for file operations. The Path that describes the file is passed in path. The how parameter specifies how the file will be opened. Because it is a varargs parameter, you can specify zero or more comma-separated arguments. (The valid values were discussed earlier and shown in Table 20-5.) If no arguments are specified, the file is opened for input operations. SeekableByteChannel is an interface that describes a channel that can be used for file operations. It is implemented by the FileChannel class. When the default file system is used, the returned object can be cast to FileChannel. You must close the channel after you have finished with it. Since all channels, including FileChannel, implement AutoCloseable, you can use a try-with-resources statement to close the file automatically instead of calling close() explicitly. This approach is used in the examples.

Next, you must obtain a buffer that will be used by the channel either by wrapping an existing array or by
allocating the buffer dynamically. The examples use allocation, but the choice is yours. Because file channels operate on byte buffers, we will use the `allocate()` method defined by `ByteBuffer` to obtain the buffer. It has this general form:

```java
static ByteBuffer allocate(int cap)
```

Here, `cap` specifies the capacity of the buffer. A reference to the buffer is returned.

After you have created the buffer, call `read()` on the channel, passing a reference to the buffer. The version of `read()` that we will use is shown next:

```java
int read(ByteBuffer buf) throws IOException
```

Each time it is called, `read()` fills the buffer specified by `buf` with data from the file. The reads are sequential, meaning that each call to `read()` reads the next buffer’s worth of bytes from the file. The `read()` method returns the number of bytes actually read. It returns `-1` when there is an attempt to read at the end of the file.

The following program puts the preceding discussion into action by reading a file called `test.txt` through a channel using explicit input operations:
Here is how the program works. First, a Path object is obtained that contains the relative path to a file called
A reference to this object is assigned to `filepath`. Next, a channel connected to the file is obtained by calling `newByteChannel()`, passing in `filepath`. Because no open option is specified, the file is opened for reading. Notice that this channel is the object managed by the `try-with-resources` statement. Thus, the channel is automatically closed when the block ends. The program then calls the `allocate()` method of `ByteBuffer` to allocate a buffer that will hold the contents of the file when it is read. A reference to this buffer is stored in `mBuf`. The contents of the file are then read, one buffer at a time, into `mBuf` through a call to `read()`. The number of bytes read is stored in `count`. Next, the buffer is rewound through a call to `rewind()`. This call is necessary because the current position is at the end of the buffer after the call to `read()`. It must be reset to the start of the buffer in order for the bytes in `mBuf` to be read by calling `get()`. (Recall that `get()` is defined by `ByteBuffer`.) Because `mBuf` is a byte buffer, the values returned by `get()` are bytes. They are cast to `char` so the file can be displayed as text. (Alternatively, it is possible to create a buffer that encodes the bytes into characters and then read that buffer.) When the end of the file has been reached, the value returned by `read()` will be –1. When this occurs, the program ends, and the channel is automatically closed.

As a point of interest, notice that the program obtains the Path within one `try` block and then uses another `try` block to obtain and manage a channel linked to that path. Although there is nothing wrong, per se, with this
approach, in many cases, it can be streamlined so that only one try block is needed. In this approach, the calls to Paths.get( ) and newByteChannel( ) are sequenced together. For example, here is a reworked version of the program that uses this approach:

```java
// A more compact way to open a channel. Requires JDK 7 or later.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;

        // Here, the channel is opened on the Path returned by Paths.get().
        // There is no need for the filepath variable.
        try (SeekableByteChannel fChan =
                Files.newByteChannel(Paths.get("test.txt")))
        {
            // Allocate a buffer.
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // Read a buffer.
                count = fChan.read(mBuf);

                // Stop when end of file is reached.
                if(count != -1) {
                    // Rewind the buffer so that it can be read.
                    mBuf.rewind();

                    // Read bytes from the buffer and show
                    // them on the screen as characters.
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);

            System.out.println();
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        }
    }
}
```
In this version, the variable `filepath` is not needed and both exceptions are handled by the same `try` statement. Because this approach is more compact, it is the approach used in the rest of the examples in this chapter. Of course, in your own code, you may encounter situations in which the creation of a `Path` object needs to be separate from the acquisition of a channel. In these cases, the previous approach can be used.

Another way to read a file is to map it to a buffer. The advantage is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file, follow this general procedure. First, obtain a `Path` object that encapsulates the file as previously described. Next, obtain a channel to that file by calling `Files.newByteChannel()` passing in the `Path` and casting the returned object to `FileChannel`. As explained, `newByteChannel()` returns a `SeekableByteChannel`. When using the default file system, this object can be cast to `FileChannel`. Then, map the channel to a buffer by calling `map()` on the channel. The `map()` method is defined by `FileChannel`. This is why the cast to `FileChannel` is needed. The `map()` function is shown here:

```
MappedByteBuffer map(FileChannel.MapMode how,
                     long pos, long size) throws
    IOException
```

The `map()` method causes the data in the file to be
mapped into a buffer in memory. The value in how determines what type of operations are allowed. It must be one of these values:

<table>
<thead>
<tr>
<th>MapMode.READ_ONLY</th>
<th>MapMode.READ_WRITE</th>
<th>MapMode.PRIVATE</th>
</tr>
</thead>
</table>

For reading a file, use MapMode.READ_ONLY. To read and write, use MapMode.READ_WRITE. MapMode.PRIVATE causes a private copy of the file to be made, and changes to the buffer do not affect the underlying file. The location within the file to begin mapping is specified by pos, and the number of bytes to map are specified by size. A reference to this buffer is returned as a MappedByteBuffer, which is a subclass of ByteBuffer. Once the file has been mapped to a buffer, you can read the file from that buffer. Here is an example that illustrates this approach:
In the program, a Path to the file is created and then opened via newByteChannel( ). The channel is cast to FileChannel and stored in fChan. Next, the size of the file is obtained by calling size( ) on the channel. Then, the entire file is mapped into memory by calling map( ) on fChan and a reference to the buffer is stored in mBuf. Notice that mBuf is declared as a reference to a MappedByteBuffer. The bytes in mBuf are read by calling get( ).

Writing to a File via a Channel
As is the case when reading from a file, there are also
several ways to write data to a file using a channel. We will begin with one of the most common. In this approach, you manually allocate a buffer, write data to that buffer, and then perform an explicit write operation to write that data to a file.

Before you can write to a file, you must open it. To do this, first obtain a Path that describes the file and then use this Path to open the file. In this example, the file will be opened for byte-based output via explicit output operations. Therefore, this example will open the file and establish a channel to it by calling Files.newByteChannel(). As shown in the previous section, the newByteChannel() method has this general form:

```java
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
    throws IOException
```

It returns a SeekableByteChannel object, which encapsulates the channel for file operations. To open a file for output, the how parameter must specify StandardOpenOption.WRITE. If you want to create the file if it does not already exist, then you must also specify StandardOpenOption.CREATE. (Other options, which are shown in Table 20-5, are also available.) As explained in the previous section, SeekableByteChannel is an interface that describes a channel that can be used for file operations. It is implemented by the FileChannel class. When the default file system is used, the return
object can be cast to FileChannel. You must close the channel after you have finished with it.

Here is one way to write to a file through a channel using explicit calls to write( ). First, obtain a Path to the file and then open it with a call to newByteChannel( ), casting the result to FileChannel. Next, allocate a byte buffer and write data to that buffer. Before the data is written to the file, call rewind( ) on the buffer to set its current position to zero. (Each output operation on the buffer increases the current position. Thus, it must be reset prior to writing to the file.) Then, call write( ) on the channel, passing in the buffer. The following program demonstrates this procedure. It writes the alphabet to a file called test.txt.
It is useful to emphasize an important aspect of this program. As mentioned, after data is written to mBuf, but before it is written to the file, a call to rewind( ) on mBuf is made. This is necessary in order to reset the current position to zero after data has been written to mBuf. Remember, each call to put( ) on mBuf advances the current position. Therefore, it is necessary for the current position to be reset to the start of the buffer before calling write( ). If this is not done, write( ) will...
think that there is no data in the buffer.

Another way to handle the resetting of the buffer between input and output operations is to call flip() instead of rewind(). The flip() method sets the value of the current position to zero and the limit to the previous current position. In the preceding example, because the capacity of the buffer equals its limit, flip() could have been used instead of rewind(). However, the two methods are not interchangeable in all cases.

In general, you must reset the buffer between read and write operations. For example, assuming the preceding example, the following loop will write the alphabet to the file three times. Pay special attention to the calls to rewind().

```java
for(int h=0; h<3; h++) {
    // Write some bytes to the buffer.
    for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

    // Rewind the buffer so that it can be written.
    mBuf.rewind();

    // Write the buffer to the output file.
    fChan.write(mBuf);

    // Rewind the buffer so that it can be written to again.
    mBuf.rewind();
}
```

Notice that rewind() is called between each read and write operation.

One other thing about the program warrants mentioning: When the buffer is written to the file, the first 26 bytes in the file will contain the output. If the file
test.txt was preexisting, then after the program executes, the first 26 bytes of test.txt will contain the alphabet, but the remainder of the file will remain unchanged.

Another way to write to a file is to map it to a buffer. The advantage to this approach is that the data written to the buffer will automatically be written to the file. No explicit write operation is necessary. To map and write the contents of a file, we will use this general procedure. First, obtain a Path object that encapsulates the file and then create a channel to that file by calling Files.newByteChannel( ), passing in the Path. Cast the reference returned by newByteChannel( ) to FileChannel. Next, map the channel to a buffer by calling map( ) on the channel. The map( ) method was described in detail in the previous section. It is summarized here for your convenience. Here is its general form:

```java
MappedByteBuffer map(FileChannel.MapMode how, 
                        long pos, long size) throws IOException
```

The map( ) method causes the data in the file to be mapped into a buffer in memory. The value in how determines what type of operations are allowed. For writing to a file, how must be MapMode.READ_WRITE. The location within the file to begin mapping is specified by pos, and the number of bytes to map are specified by size. A reference to this buffer is returned. Once the file has been mapped to a buffer, you can write data to that buffer, and it will automatically be written
to the file. Therefore, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used. Notice that in the call to `newByteChannel()`, the open option `StandardOpenOption.READ` has been added. This is because a mapped buffer can either be read-only or read/write. Thus, to write to the mapped buffer, the channel must be opened as read/write.

```java
// Write to a mapped file. Requires JDK 7 or later.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelWrite {
    public static void main(String args[]) {
        // Obtain a channel to a file within a try-with-resources block.
        try (FileChannel fChan = (FileChannel)
             Files.newByteChannel(Paths.get("test.txt"),
                                 StandardOpenOption.WRITE,
                                 StandardOpenOption.READ,
                                 StandardOpenOption.CREATE)) {

            // Then, map the file into a buffer.
            MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));
        } catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        } catch (IOException e) {
            System.out.println("I/O Error " + e);
        }
    }
}
```

As you can see, there are no explicit write operations to the channel itself. Because `mBuf` is mapped to the file,
changes to mBuf are automatically reflected in the underlying file.

Copying a File Using NIO

NIO simplifies several types of file operations. Although we can’t examine them all, an example will give you an idea of what is available. The following program copies a file using a call to a single NIO method: copy( ), which is a static method defined by Files. It has several forms. Here is the one we will be using:

```java
static Path copy(Path src, Path dest, CopyOption ... how) throws IOException
```

The file specified by src is copied to the file specified by dest. How the copy is performed is specified by how. Because it is a varargs parameter, it can be missing. If specified, it can be one or more of these values, which are valid for all file systems:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StandardCopyOption.COPY_ATTRIBUTES</td>
<td>Request that the file’s attributes be copied.</td>
</tr>
<tr>
<td>StandardLinkOption.NOFOLLOW_LINKS</td>
<td>Do not follow symbolic links.</td>
</tr>
<tr>
<td>StandardCopyOption.REPLACE_EXISTING</td>
<td>Overwrite a preexisting file.</td>
</tr>
</tbody>
</table>

Other options may be supported, depending on the implementation.

The following program demonstrates copy( ). The source and destination files are specified on the command line, with the source file specified first. Notice how short the program is. You might want to compare this version of the file copy program to the one found in Chapter 13. As you will find, the part of the program that actually copies the file is substantially shorter in the
Use NIO for Stream-Based I/O

Beginning with JDK 7, you can use NIO to open an I/O stream. Once you have a Path, open a file by calling ```newInputStream()``` or ```newOutputStream()```), which are static methods defined by Files. These methods return a stream connected to the specified file. In either case, the stream can then be operated on in the way described in Chapter 19, and the same techniques apply. The advantage of using Path to open a file is that all of the features defined by NIO are available for your use.

To open a file for stream-based input, use

```java
// Copy a file using NIO. Requires JDK 7 or later.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

    public static void main(String args[]) {

        if(args.length != 2) {
            System.out.println("Usage: Copy from to");
            return;
        }

        Path source = Paths.get(args[0]);
        Path target = Paths.get(args[1]);

        // Copy the file.
        Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);

        } catch(InvalidPathException e) {
            System.out.println("Path Error "+ e);
        } catch (IOException e) {
            System.out.println("I/O Error "+ e);
        }
    }
}
```
Files.newInputStream(). It is shown here:

```java
static InputStream newInputStream(Path path,
   OpenOption … how)
   throws IOException
```

Here, path specifies the file to open and how specifies how the file will be opened. It must be one or more of the values defined by StandardOpenOption, described earlier. (Of course, only those options that relate to an input stream will apply.) If no options are specified, then the file is opened as if StandardOpenOption.READ were passed.

Once opened, you can use any of the methods defined by InputStream. For example, you can use read() to read bytes from the file.

The following program demonstrates the use of NIO-based stream I/O. It reworks the ShowFile program from Chapter 13 so that it uses NIO features to open the file and obtain a stream. As you can see, it is very similar to the original, except for the use of Path and newInputStream().
Because the stream returned by `newInputStream()` is a normal stream, it can be used like any other stream. For example, you can wrap the stream inside a buffered stream, such as a `BufferedInputStream`, to provide buffering, as shown here:

```java
new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))
```

Now, all reads will be automatically buffered.
To open a file for output, use `Files.newOutputStream()`. It is shown here:

```java
static OutputStream newOutputStream(Path path, OpenOption ... how)
        throws IOException
```

Here, `path` specifies the file to open and `how` specifies how the file will be opened. It must be one or more of the values defined by `StandardOpenOption`, described earlier. (Of course, only those options that relate to an output stream will apply.) If no options are specified, then the file is opened as if `StandardOpenOption.WRITE`, `StandardOpenOption.CREATE`, and `StandardOpenOption.TRUNCATE_EXISTING` were passed.

The methodology for using `newOutputStream()` is similar to that shown previously for `newInputStream()`. Once opened, you can use any of the methods defined by `OutputStream`. For example, you can use `write()` to write bytes to the file. You can also wrap the stream inside a `BufferedOutputStream` to buffer the stream.

The following program shows `newOutputStream()` in action. It writes the alphabet to a file called `test.txt`. Notice the use of buffered I/O.
Use NIO for Path and File System Operations

At the beginning of Chapter 19, the File class in the java.io package was examined. As explained there, the File class deals with the file system and with the various attributes associated with a file, such as whether a file is read-only, hidden, and so on. It was also used to obtain information about a file’s path. With the advent of JDK 7, the interfaces and classes defined by NIO.2 offer a better way to perform these functions. The benefits include support for symbolic links, better support for directory tree traversal, and improved handling of metadata, among others. The following sections show samples of two common file system operations: obtaining information about a path and file and getting the contents of a directory.

```java
import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
    public static void main(String args[])
    {
        // Open the file and obtain a stream linked to it.
        try (OutputStream fout =
                new BufferedOutputStream(Files.newOutputStream(Paths.get("test.txt"))))
        {
            // Write some bytes to the stream.
            for(int i=0; i < 26; i++)
                fout.write((byte)('A' + i));
        }
        catch(InvalidPathException e) {
            System.out.println("Path Error " + e);
        }
        catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
```
REMEMBER

If you want to update older code that uses java.io.File to the new Path interface, you can use the toPath( ) method to obtain a Path instance from a File instance. The toPath( ) method was added to File by JDK 7.

Obtain Information About a Path and a File

Information about a path can be obtained by using methods defined by Path. Some attributes associated with the file described by a Path (such as whether or not the file is hidden) are obtained by using methods defined by Files. The Path methods used here are getName( ), getParent( ), and toAbsolutePath( ). Those provided by Files are isExecutable( ), isHidden( ), isReadable( ), isWritable( ), and exists( ). These are summarized in Tables 20-3 and 20-4, shown earlier.

CAUTION

Methods such as isExecutable( ), isReadable( ), isWritable( ), and exists( ) must be used with care because the state of the file system may change after the call, in which case a program malfunction could occur. Such a situation could have security implications.

Other file attributes are obtained by requesting a list of attributes by calling Files.readAttributes( ). In the program, this method is called to obtain the BasicFileAttributes associated with a file, but the general
approach applies to other types of attributes.

The following program demonstrates several of the Path and Files methods, along with several methods provided by BasicFileAttributes. This program assumes that a file called test.txt exists in a directory called examples, which must be a subdirectory of the current directory.

```java
// Obtain information about a path and a file.
// Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
    public static void main(String args[]) {
        Path filepath = Paths.get("examples\test.txt");
        System.out.println("File Name: " + filepath.getName(1));
        System.out.println("Path: " + filepath);
        System.out.println("Absolute Path: " + filepath.toAbsolutePath());
        System.out.println("Parent: " + filepath.getParent());

        if(Files.exists(filepath))
            System.out.println("File exists");
        else
            System.out.println("File does not exist");

        try {
            if(Files.isHidden(filepath))
                System.out.println("File is hidden");
            else
                System.out.println("File is not hidden");
        } catch(IOException e) {
            System.out.println("I/O Error: " + e);
        }
    }
}
If you execute this program from a directory called MyDir, which has a subdirectory called examples, and the examples directory contains the test.txt file, then you will see output similar to that shown here. (Of course, file size and time will differ.)

File Name: test.txt
Path: examples\test.txt
Absolute Path: C:\MyDir\examples\test.txt
Parent: examples
File exists
File is not hidden
File is writable
File is readable
The file is not a directory
The file is a normal file
The file is not a symbolic link
File last modified: 2010-09-01T18:20:46.380445Z
File size: 18 Bytes

If you are using a computer that supports the FAT file system (i.e., the DOS file system), then you might want to try using the methods defined by DosFileAttributes. If you are using a POSIX-compatible system, then try using PosixFileAttributes.

List the Contents of a Directory
If a path describes a directory, then you can read the contents of that directory by using static methods defined by Files. To do this, you first obtain a directory stream by calling newDirectoryStream(), passing in a Path that describes the directory. One form of newDirectoryStream() is shown here:

```java
static DirectoryStream<Path>
    newDirectoryStream(Path dirPath)
    throws IOException
```

Here, dirPath encapsulates the path to the directory. The method returns a DirectoryStream<Path> object that can be used to obtain the contents of the directory. It will throw an IOException if an I/O error occurs and a NotDirectoryException (which is a subclass of IOException) if the specified path is not a directory. A SecurityException is also possible if access to the directory is not permitted.

DirectoryStream<Path> implements AutoCloseable, so it can be managed by a try-with-resources statement. It
also implements `Iterable<Path>`.

This means that you can obtain the contents of the directory by iterating over `DirectoryStream` object. When iterating, each directory entry is represented by a `Path` instance. An easy way to iterate over a `DirectoryStream` is to use a for-each style `for` `loop`. It is important to understand, however, that the iterator implemented by `DirectoryStream<Path>` can be obtained only once for each instance. Thus, the `iterator()` method can be called only once, and a for-each `loop` can be executed only once.

The following program displays the contents of a directory called `MyDir`:
Here is sample output from the program:

Directory of \MyDir
   DirList.class
   DirList.java
   <DIR> examples
       Test.txt

You can filter the contents of a directory in two ways.
The easiest is to use this version of newDirectoryStream( ):
In this version, only files that match the wildcard filename specified by wildcard will be obtained. For wildcard, you can specify either a complete filename or a glob. A glob is a string that defines a general pattern that will match one or more files using the familiar * and ? wildcard characters. These match zero or more of any character and any one character, respectively. The following are also recognized within a glob.

<table>
<thead>
<tr>
<th>**</th>
<th>Matches zero or more of any character across directories.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[chars]</td>
<td>Matches any one character in chars. A * or ? within chars will be treated as a normal character, not a wildcard. A range can be specified by use of a hyphen, such as [x-z].</td>
</tr>
<tr>
<td>{globlist}</td>
<td>Matches any one of the globs specified in a comma-separated list of globs in globlist.</td>
</tr>
</tbody>
</table>

You can specify a * or ? character, using \* and \?. To specify a \, use \\.. You can experiment with a glob by substituting this call to newDirectoryStream( ) into the previous program:

```java
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

This obtains a directory stream that contains only those files whose names begin with either “Path” or “Dir” and use either the “java” or “class” extension. Thus, it would match names like DirList.java and PathDemo.java, but not MyPathDemo.java, for example.

Another way to filter a directory is to use this version of newDirectoryStream( ):

```java
static DirectoryStream<Path>
newDirectoryStream(Path dirPath, String wildcard)
throws IOException
```

You can specify a * or ? character, using \* and \?. To specify a \, use \\.. You can experiment with a glob by substituting this call to newDirectoryStream( ) into the previous program:

```java
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```
newDirectoryStream(Path dirPath,
     DirectoryStream.Filter<? super Path>
           filefilter)
          throws IOException

Here, DirectoryStream.Filter is an interface that specifies
the following method:

    boolean accept(T entry) throws IOException

In this case, T will be Path. If you want to include entry
in the list, return true. Otherwise, return false. This form
of newDirectoryStream() offers the advantage of being
able to filter a directory based on something other than a
filename. For example, you can filter based on size,
creation date, modification date, or attribute, to name a
few.

The following program demonstrates the process. It
will list only those files that are writable.

// Display a directory of only those files that are writable.
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        // Create a filter that returns true only for writable files.
        DirectoryStream.Filter<Path> how = new DirectoryStream.Filter<Path>() {
            public boolean accept(Path filename) throws IOException {
                if(Files.isWritable(filename)) return true;
            return false;
        };
    }
Use walkFileTree() to List a Directory Tree

The preceding examples have obtained the contents of only a single directory. However, sometimes you will want to obtain a list of the files in a directory tree. In the past, this was quite a chore, but NIO.2 makes it easy because now you can use the walkFileTree() method defined by Files to process a directory tree. It has two forms. The one used in this chapter is shown here:

```java
static Path walkFileTree(Path root, FileVisitor<? extends Path> fv) throws IOException
```

The path to the starting point of the directory walk is passed in `root`. An instance of `FileVisitor` is passed in `fv`. The implementation of `FileVisitor` determines how the
directory tree is traversed, and it gives you access to the
directory information. If an I/O error occurs, an
IOException is thrown. A SecurityException is also
possible.

FileVisitor is an interface that defines how files are
visited when a directory tree is traversed. It is a generic
interface that is declared like this:

```java
interface FileVisitor<T>
```

For use in walkFileTree(), T will be Path (or any type
derived from Path). FileVisitor defines the following
methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>visitFile(T file, BasicFileAttributes attrs)</code></td>
<td>Called when a file is visited. The file is passed in <code>file</code>, and the attributes associated with the file are passed in <code>attrs</code>. The result is returned.</td>
</tr>
<tr>
<td><code>visitFileFailed(T file, IOException exc)</code></td>
<td>Called when an attempt to visit a file fails. The file that failed is passed in <code>file</code>, and the IOException is passed in <code>exc</code>. The result is returned.</td>
</tr>
</tbody>
</table>

Notice that each method returns a FileVisitResult. This
enumeration defines the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTINUE</td>
<td></td>
</tr>
<tr>
<td>SKIP_SIBLINGS</td>
<td></td>
</tr>
<tr>
<td>SKIP_SUBTREE</td>
<td></td>
</tr>
<tr>
<td>TERMINATE</td>
<td></td>
</tr>
</tbody>
</table>

In general, to continue traversing the directory and
subdirectories, a method should return CONTINUE. For
preVisitDirectory(), return SKIP_SIBLINGS to bypass the
directory and its siblings and prevent postVisitDirectory(
from being called. To bypass just the directory and subdirectories, return SKIP_SUBTREE. To stop the directory traversal, return TERMINATE.

Although it is certainly possible to create your own visitor class that implements these methods defined by FileVisitor, you won’t normally do so because a simple implementation is provided by SimpleFileVisitor. You can just override the default implementation of the method or methods in which you are interested. Here is a short example that illustrates the process. It displays all files in the directory tree that has `\MyDir` as its root. Notice how short this program is.
Here is sample output produced by the program when used on the same MyDir directory shown earlier. In this example, the subdirectory called examples contains one file called MyProgram.java.

Directory tree starting with \MyDir:

\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt

In the program, the class MyFileVisitor extends SimpleFileVisitor, overriding only the visitFile( ) method. In this example, visitFile( ) simply displays the
files, but more sophisticated functionality is easy to achieve. For example, you could filter the files or perform actions on the files, such as copying them to a backup device. For the sake of clarity, a named class was used to override visitFile(), but you could also use an anonymous inner class.

One last point: It is possible to watch a directory for changes by using java.nio.file.WatchService.

Pre-JDK 7 Channel-Based Examples
Before concluding this chapter, one more aspect of NIO needs to be covered. The preceding sections have used several of the new features added to NIO by JDK 7. However, there is still much pre-JDK 7 code in existence that will need to be maintained or possibly converted to use the new features. For this reason, the following sections show how to read and write files using the pre-JDK 7 NIO system. They do so by reworking some of the examples shown earlier so that they use the original NIO features, rather than the new JDK 7 (NIO.2) features. This means that the examples in this section will work with versions of Java prior to JDK 7.

The key difference between pre-JDK 7 NIO code and new NIO code is the Path interface, which was added by JDK 7. Thus, pre-JDK 7 code does not use Path to describe a file or open a channel to it. Also, pre-JDK 7 code does not use try-with-resource statements since automatic resource management was also added by JDK 7.
The examples in this section describe how legacy NIO code works. This section is strictly for the benefit of those programmers working on pre-JDK 7 code or using pre-JDK 7 compilers. New code should take advantage of the NIO features added by JDK 7.

Read a File, Pre-JDK 7

This section reworks the two channel-based file input examples shown earlier so they use only pre-JDK 7 features. The first example reads a file by manually allocating a buffer and then performing an explicit read operation. The second example uses a mapped file, which automates the process.

When using a pre-JDK 7 version of Java to read a file using a channel and a manually allocated buffer, you first open the file for input using FileInputStream, using the same mechanism explained in Chapter 19. Next, obtain a channel to this file by calling getChannel() on the FileInputStream object. It has this general form:

```
FileChannel getChannel()
```

It returns a FileChannel object, which encapsulates the channel for file operations. Then, call allocate() to allocate a buffer. Because file channels operate on byte buffers, you will use the allocate() method defined by ByteBuffer, which works as previously described.

The following program shows how to read and
display a file called test.txt through a channel using explicit input operations for versions of Java prior to JDK 7:

```java
// Use Channels to read a file. Pre-JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;
        int count;

        try {
            // First, open a file for input.
            fIn = new FileInputStream("test.txt");

            // Next, obtain a channel to that file.
            fChan = fIn.getChannel();

            // Allocate a buffer.
            mBuf = ByteBuffer.allocate(128);

            do {
                // Read a buffer.
                count = fChan.read(mBuf);

                // Stop when end of file is reached.
                if(count != -1) {
                    // Rewind the buffer so that it can be read.
                    mBuf.rewind();

                    // Read bytes from the buffer and show
                    // them on the screen.
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);
        }
    }
}
```
In this program, notice that the file is opened by using the FileInputStream constructor, and a reference to that object is assigned to fIn. Next, a channel connected to the file is obtained by calling getChannel( ) on fIn. After this point, the program works like the JDK 7 version shown previously. To synopsize: The program then calls the allocate( ) method of ByteBuffer to allocate a buffer that will hold the contents of the file when it is read. A byte buffer is used because FileChannel operates on bytes. A reference to this buffer is stored in mBuf. The contents of the file are then read, one buffer at a time, into mBuf through a call to read( ). The number of bytes read is stored in count. Next, the buffer is rewound through a call to rewind( ). This call is necessary because the current position is at the end of the buffer after the call to read( ), and it must be reset to the start of the buffer in order for the bytes in mBuf to be read by calling get( ). When the end of the file has been reached, the value returned by read( ) will be –1. When this occurs, the
program ends, explicitly closing the channel and the file.

Another way to read a file is to map it to a buffer. As explained earlier, a principal advantage to this approach is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file using pre-JDK 7 NIO, first open the file using FileInputStream. Next, obtain a channel to that file by calling getChannel( ) on the file object. Then, map the channel to a buffer by calling map( ) on the FileChannel object. The map( ) method works as described earlier.

The following program reworks the preceding example so that it uses only pre-JDK 7 features to create a mapped file:

```java
// Use a mapped file to read a file. Pre-JDK 7 version.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
        FileChannel fChan = null;
        long fSize;
        MappedByteBuffer mBuf;

        try {
            // First, open a file for input.
            fIn = new FileInputStream("test.txt");

            // Next, obtain a channel to that file.
            fChan = fIn.getChannel();

            // Get the size of the file.
            fSize = fChan.size();
```
In the program, the file is opened by using the FileInputStream constructor, and a reference to that object is assigned to fIn. A channel connected to the file is obtained by calling getChannel( ) on fIn. Next, the size of the file is obtained. Then, the entire file is mapped into memory by calling map( ), and a reference to the buffer is stored in mBuf. The bytes in mBuf are read by calling get( ).

Write to a File, Pre-JDK 7

This section reworks the two channel-based file output examples shown earlier so that they use only pre-JDK 7 features. The first example writes to a file by manually allocating a buffer and then performing an explicit output operation. The second example uses a mapped file, which automates the process. In both cases, neither
Path nor try-with-resources is used. This is because neither were part of Java until JDK 7.

When using a pre-JDK 7 version of Java to write a file using a channel and a manually allocated buffer, first open the file for output. This is done by creating a FileOutputStream, as described in Chapter 19. Next, obtain a channel to the file by calling getChannel() and then allocate a byte buffer by calling allocate(), as described in the previous section. Next, put the data you want to write into that buffer, and then call write() on the channel. The following program demonstrates this procedure. It writes the alphabet to a file called test.txt.

```java
// Write to a file using NIO. Pre-JDK 7 Version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
    public static void main(String args[]) {
        FileOutputStream fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            // First, open the output file.
            fOut = new FileOutputStream("test.txt");

            // Next, get a channel to the output file.
            fChan = fOut.getChannel();

            // Create a buffer.
            mBuf = ByteBuffer.allocate(26);

            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // Rewind the buffer so that it can be written.
            mBuf.rewind();
        }
    }
}
```
The call to `rewind()` on `mBuf` is necessary in order to reset the current position to zero after data has been written to `mBuf`. Remember, each call to `put()` advances the current position. Therefore, it is necessary for the current position to be reset to the start of the buffer before calling `write()`. If this is not done, `write()` will think that there is no data in the buffer.

When using a pre-JDK 7 version of Java to write to a file using a mapped file, follow these steps. First, open the file for read/write operations by creating a `RandomAccessFile` object. This is necessary to enable the file to be both read from and written to. Next, map that file to a buffer by calling `map()` on that object. Then, write to the buffer. Because the buffer is mapped to the file, any changes to that buffer are automatically reflected in the file. Thus, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a
A mapped file is used:

```java
// Write to a mapped file. Pre JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
    public static void main(String args[]) {
        RandomAccessFile fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            fOut = new RandomAccessFile("test.txt", "rw");

            // Next, obtain a channel to that file.
            fChan = fOut.getChannel();

            // Then, map the file into a buffer.
            mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // Write some bytes to the buffer.
            for(int i=0; i<26; i++)
                mBuf.put((byte)'A' + i);
        }
        catch (IOException e) {
            System.out.println("I/O Error " + e);
        }
        finally {
            try {
                if(fChan != null) fChan.close(); // close channel
            }
            catch (IOException e) {
                System.out.println("Error Closing Channel.");
            }
            try {
                if(fOut != null) fOut.close(); // close file
            }
            catch (IOException e) {
                System.out.println("Error Closing File.");
            }
        }
    }
}
```

As you can see, there are no explicit write operations to the channel itself. Because `mBuf` is mapped to the file, changes to `mBuf` are automatically reflected in the underlying file.
As all readers know, Java is practically a synonym for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross-platform, portable code. However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the java.net package. They provide an easy-to-use means by which programmers of all skill levels can access network resources.

This chapter explores the java.net package. It is important to emphasize that networking is a very large and at times complicated topic. It is not possible for this book to discuss all of the capabilities contained in java.net. Instead, this chapter focuses on several of its core classes and interfaces.

Networking Basics
Before we begin, it will be useful to review some key networking concepts and terms. At the core of Java’s networking support is the concept of a socket. A socket identifies an endpoint in a network. The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term Berkeley socket is also used. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a port, which is a numbered socket on a particular machine. A server
process is said to “listen” to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. Transmission Control Protocol (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data. A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for whois; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. Here’s how it works. When a client requests a file
from an HTTP server, an action known as a hit, it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not the request can be fulfilled and why.

A key component of the Internet is the address. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, when using Java, you won’t normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you.

Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its domain name, describes a machine’s location in a name space. For example, www.HerbSchildt.com is in the COM top-level domain (reserved for U.S. commercial sites); it is called HerbSchildt, and www identifies the server for web requests. An Internet domain name is mapped to an IP address by the Domain Naming Service (DNS). This enables users to work with domain names, but the Internet operates on IP addresses.
The Networking Classes and Interfaces
Java supports TCP/IP both by extending the already established stream I/O interface introduced in Chapter 19 and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the java.net package are shown here:

<table>
<thead>
<tr>
<th>Authenticator</th>
<th>Inet6Address</th>
<th>ServerSocket</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheRequest</td>
<td>InetAddress</td>
<td>Socket</td>
</tr>
<tr>
<td>CacheResponse</td>
<td>InetSocketAddress</td>
<td>SocketAddress</td>
</tr>
<tr>
<td>ContentHandler</td>
<td>InterfaceAddress</td>
<td>SocketImpl</td>
</tr>
<tr>
<td>CookieHandler</td>
<td>JarURLConnection</td>
<td>SocketPermission</td>
</tr>
<tr>
<td>CookieManager</td>
<td>MulticastSocket</td>
<td>StandardSocketOption (Added by JDK 7.)</td>
</tr>
<tr>
<td>DatagramPacket</td>
<td>NetPermission</td>
<td>URI</td>
</tr>
<tr>
<td>DatagramSocket</td>
<td>NetworkInterface</td>
<td>URL</td>
</tr>
<tr>
<td>DatagramSocketImpl</td>
<td>PasswordAuthentication</td>
<td>URLClassLoader</td>
</tr>
<tr>
<td>HttpURLConnection</td>
<td>Proxy</td>
<td>URLConnection</td>
</tr>
<tr>
<td>HttpURLConnection</td>
<td>ProxySelector</td>
<td>URLEncoder</td>
</tr>
<tr>
<td>IDN</td>
<td>ResponseCache</td>
<td>URLStreamHandler</td>
</tr>
<tr>
<td>Inet4Address</td>
<td>SecureCacheResponse</td>
<td>URLStreamHandler</td>
</tr>
</tbody>
</table>

The java.net package’s interfaces are listed here:

<table>
<thead>
<tr>
<th>ContentHandlerFactory</th>
<th>FileNameMap</th>
<th>SocketOptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CookiePolicy</td>
<td>ProtocolFamily (Added by JDK 7.)</td>
<td>URLStreamHandlerFactory</td>
</tr>
<tr>
<td>CookieStore</td>
<td>SocketImplFactory</td>
<td></td>
</tr>
<tr>
<td>DatagramSocketImplFactory</td>
<td>SocketOption (Added by JDK 7.)</td>
<td></td>
</tr>
</tbody>
</table>

In the sections that follow, we will examine the main networking classes and show several examples that apply to them. Once you understand these core networking classes, you will be able to easily explore the others on your own.
InetAddress
The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. InetAddress can handle both IPv4 and IPv6 addresses.

Factory Methods
The InetAddress class has no visible constructors. To create an InetAddress object, you have to use one of the available factory methods. Factory methods are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used InetAddress factory methods are shown here:

```java
static InetAddress getLocalHost( )
    throws UnknownHostException

static InetAddress getByName(String hostName)
    throws UnknownHostException

static InetAddress[ ] getAllByName(String hostName)
    throws UnknownHostException
```

The getLocalHost( ) method simply returns the InetAddress object that represents the local host. The getByName( ) method returns an InetAddress for a host name passed to it. If these methods are unable to resolve
the host name, they throw an UnknownHostException.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The `getAllByName()` factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException if it can’t resolve the name to at least one address.

`InetAddress` also includes the factory method `getByAddress()`, which takes an IP address and returns an `InetAddress` object. Either an IPv4 or an IPv6 address can be used.

The following example prints the addresses and names of the local machine and two Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
    public static void main(String args[]) throws UnknownHostException {
        InetAddress Address = InetAddress.getLocalHost();
        System.out.println(Address);

        Address = InetAddress.getByName("www.HerbSchildt.com");
        System.out.println(Address);

        InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
        for (int i=0; i<SW.length; i++)
            System.out.println(SW[i]);
    }
}
```

Here is the output produced by this program. (Of course, the output you see may be slightly different.)

default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/216.66.31.161
Instance Methods
The InetAddress class has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the more commonly used methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(Object other)</td>
<td>Returns true if this object has the same Internet address as other.</td>
</tr>
<tr>
<td>byte[ ] getAddress( )</td>
<td>Returns a byte array that represents the object’s IP address in network byte order.</td>
</tr>
<tr>
<td>String getHostAddress( )</td>
<td>Returns a string that represents the host address associated with the InetAddress object.</td>
</tr>
<tr>
<td>String getHostName( )</td>
<td>Returns a string that represents the host name associated with the InetAddress object.</td>
</tr>
<tr>
<td>boolean isMulticastAddress( )</td>
<td>Returns true if this address is a multicast address. Otherwise, it returns false.</td>
</tr>
<tr>
<td>String toString( )</td>
<td>Returns a string that lists the host name and the IP address for convenience.</td>
</tr>
</tbody>
</table>

Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn’t have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server. This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly.

Inet4Address and Inet6Address
Beginning with version 1.4, Java has included support for IPv6 addresses. Because of this, two subclasses of InetAddress were created: Inet4Address and Inet6Address. Inet4Address represents a traditional-style
IPv4 address. Inet6Address encapsulates a new-style IPv6 address. Because they are subclasses of InetAddress, an InetAddress reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use InetAddress when working with IP addresses because it can accommodate both styles.

TCP/IP Client Sockets
TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java’s I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

NOTE
As a general rule, applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a “listener,” which waits for clients to connect before doing anything. Thus, ServerSocket is for servers. The Socket class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined
The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Socket(String hostName, int port)</code></td>
<td>Creates a socket connected to the named host and port.</td>
</tr>
<tr>
<td><code>throws UnknownHostException, IOException</code></td>
<td></td>
</tr>
<tr>
<td><code>Socket(InetAddress ipAddress, int port)</code></td>
<td>Creates a socket using a preexisting InetAddress object and a port.</td>
</tr>
<tr>
<td><code>throws IOException</code></td>
<td></td>
</tr>
</tbody>
</table>

Socket defines several instance methods. For example, a Socket can be examined at any time for the address and port information associated with it, by use of the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InetAddress getInetAddress()</code></td>
<td>Returns the InetAddress associated with the Socket object. It returns null if the socket is not connected.</td>
</tr>
<tr>
<td><code>int getPort()</code></td>
<td>Returns the remote port to which the invoking Socket object is connected. It returns 0 if the socket is not connected.</td>
</tr>
<tr>
<td><code>int getLocalPort()</code></td>
<td>Returns the local port to which the invoking Socket object is bound. It returns -1 if the socket is not bound.</td>
</tr>
</tbody>
</table>

You can gain access to the input and output streams associated with a Socket by use of the `getInputStream()` and `getOutputStream()` methods, as shown here. Each can throw an IOException if the socket has been invalidated by a loss of connection. These streams are used exactly like the I/O streams described in Chapter 19 to send and receive data.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InputStream getInputStream()</code></td>
<td>Returns the InputStream associated with the invoking socket.</td>
</tr>
<tr>
<td><code>throws IOException</code></td>
<td></td>
</tr>
<tr>
<td><code>OutputStream getOutputStream()</code></td>
<td>Returns the OutputStream associated with the invoking socket.</td>
</tr>
<tr>
<td><code>throws IOException</code></td>
<td></td>
</tr>
</tbody>
</table>

Several other methods are available, including `connect()`.
which allows you to specify a new connection; isConnected( ), which returns true if the socket is connected to a server; isBound( ), which returns true if the socket is bound to an address; and isClosed( ), which returns true if the socket is closed. To close a socket, call close( ). Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, Socket also implements AutoCloseable, which means that you can use a try-with-resources block to manage a socket.

The following program provides a simple Socket example. It opens a connection to a “whois” port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, and then send back the IP address and contact information for that site.
If, for example, you obtained information about MHProfessional.com, you’d get something similar to the following:

Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered with many different competing registrars. Go to http://www.internic.net for detailed information.

Domain Name: MHPROFESSIONAL.COM
Registrar: MELBOURNE IT, LTD. D/B/A INTERNET NAMES WORLDWIDE
Whois Server: whois.melbourneit.com
Referral URL: http://www.melbourneit.com
Name Server: NS1.MHEDU.COM
Name Server: NS2.MHEDU.COM

Here is how the program works. First, a Socket is
constructed that specifies the host name “whois.internic.net” and the port number 43. Internic.net is the InterNIC web site that handles whois requests. Port 43 is the whois port. Next, both input and output streams are opened on the socket. Then, a string is constructed that contains the name of the web site you want to obtain information about. In this case, if no web site is specified on the command line, then “MHProfessional.com” is used. The string is converted into a byte array and then sent out of the socket. The response is read by inputting from the socket, and the results are displayed. Finally, the socket is closed, which also closes the I/O streams.

In the preceding example, the socket was closed manually by calling close( ). If you are using JDK 7 or later, then you can use a try-with-resources block to automatically close the socket. For example, here is another way to write the main( ) method of the previous program:

```java
// Use try-with-resources to close a socket.
public static void main(String args[]) throws Exception {
    int c;

    // Create a socket connected to internic.net, port 43. Manage this
    // socket with a try-with-resources block.
    try (Socket s = new Socket("whois.internic.net", 43)) {

        // Obtain input and output streams.
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();

        // Construct a request string.
        String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
        // Convert to bytes.
        byte buf[] = str.getBytes();
        // Send request.
        out.write(buf);
```
In this version, the socket is automatically closed when the try block ends.

So the examples will work with versions of Java prior to JDK 7, and to clearly illustrate when a network resource can be closed, subsequent examples will continue to call close() explicitly. However, in your own code, you should consider using automatic resource management since it offers a more streamlined approach. One other point: In this version, exceptions are still thrown out of main(), but they could be handled by adding catch clauses to the end of the try-with-resources block.

NOTE

For simplicity, the examples in this chapter simply throw all exceptions out of main(). This allows the logic of the network code to be clearly illustrated. However, in real-world code, you will normally need to handle the exceptions in an appropriate way.

URL

The preceding example was rather obscure because the modern Internet is not about the older protocols such as whois, finger, and FTP. It is about WWW, the World Wide Web. The Web is a loose collection of higher-level
protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scalable way to locate all of the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. Within Java’s network class library, the URL class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples: http://www.MHProfessional.com/ and http://www.MHProfessional.com:80/index.htm. A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the “http://” from your URL specification). The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (/) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, “:80” is redundant.) The fourth part is the
actual file path. Most HTTP servers will append a file named index.html or index.htm to URLs that refer directly to a directory resource. Thus, http://www.MHProfessional.com/ is the same as http://www.MHProfessional.com/index.htm.

Java’s URL class has several constructors; each can throw a MalformedURLException. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

```java
URL(String urlSpecifier) throws MalformedURLException
```

The next two forms of the constructor allow you to break up the URL into its component parts:

```java
URL(String protocolName, String hostName, int port, String path)
    throws MalformedURLException

URL(String protocolName, String hostName, String path)
    throws MalformedURLException
```

Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it’s really quite easy and useful.

```java
URL(URL urlObj, String urlSpecifier) throws MalformedURLException
```

The following example creates a URL to HerbSchildt.com’s articles page and then examines its
import java.net.*;
class URLDemo {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://www.HerbSchildt.com/Articles");

        System.out.println("Protocol: " + hp.getProtocol());
        System.out.println("Port: " + hp.getPort());

        System.out.println("Host: " + hp.getHost());
        System.out.println("File: " + hp.getFile());
        System.out.println("Ext:" + hp.toExternalForm());
    }
}

When you run this, you will get the following output:

Protocol: http
Port: -1
Host: www.HerbSchildt.com
File: /Articles
Ext:http://www.HerbSchildt.com/Articles

Notice that the port is -1; this means that a port was not explicitly set. Given a URL object, you can retrieve the data associated with it. To access the actual bits or content information of a URL, create a URLConnection object from it, using its openConnection( ) method, like this:

    urlc = url.openConnection()

openConnection( ) has the following general form:

    URLConnection openConnection( ) throws IOException

It returns a URLConnection object associated with the invoking URL object. Notice that it may throw an IOException.
URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for URL objects that are using the HTTP protocol.

URLConnection defines several methods. Here is a sampling:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getContentLength( )</td>
<td>Returns the size in bytes of the content associated with the resource. If the length is unavailable, -1 is returned.</td>
</tr>
<tr>
<td>long getContentLengthLong( )</td>
<td>Returns the size in bytes of the content associated with the resource. If the length is unavailable, -1 is returned. (Added by JDK 7.)</td>
</tr>
<tr>
<td>String getContentType( )</td>
<td>Returns the type of content found in the resource. This is the value of the content-type header field. Returns null if the content type is not available.</td>
</tr>
<tr>
<td>long getDate( )</td>
<td>Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT.</td>
</tr>
<tr>
<td>long getExpiration( )</td>
<td>Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable.</td>
</tr>
</tbody>
</table>
Notice that URLConnection defines several methods that handle header information. A header consists of pairs of keys and values represented as strings. By using `getHeaderField(int idx)`, you can obtain the value associated with a header key. By calling `getHeaderFields()`, you can obtain a map that contains all of the headers. Several standard header fields are available directly through methods such as `getDate()` and `getContentType()`.

The following example creates a URLConnection using the `openConnection()` method of a URL object and then uses it to examine the document's properties and content:
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
    public static void main(String args[]) throws Exception {
        int c;
        URL hp = new URL("http://www.internic.net");
        URLConnection hpCon = hp.openConnection();

        // get date
        long d = hpCon.getDate();
        if(d==0)
            System.out.println("No date information.");
        else
            System.out.println("Date: " + new Date(d));

        // get content type
        System.out.println("Content-Type: " + hpCon.getContentType());

        // get expiration date
        d = hpCon.getExpiration();
        if(d==0)
            System.out.println("No expiration information.");
        else
            System.out.println("Expires: " + new Date(d));

        // get last-modified date
        d = hpCon.getLastModified();
        if(d==0)
            System.out.println("No last-modified information.");
        else
            System.out.println("Last-Modified: " + new Date(d));
    }
}
The program establishes an HTTP connection to www.internic.net over port 80. It then displays several header values and retrieves the content. Here are the first lines of the output (the precise output will vary over time).

```
Date: Mon Oct 04 15:53:24 CDT 2010
Content-Type: text/html; charset=UTF-8
No expiration information.
Last-Modified: Thu Sep 24 15:22:52 CDT 2009
Content-Length: 7316
=== Content ===
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>InterNIC | The Internet's Network Information Center</title>
.
.
.

HttpURLConnection
```
Java provides a subclass of URLConnection that provides support for HTTP connections. This class is called HttpURLConnection. You obtain an HttpURLConnection in the same way just shown, by calling openConnection(
on a URL object, but you must cast the result to HttpURLConnection. (Of course, you must make sure that you are actually opening an HTTP connection.) Once you have obtained a reference to an HttpURLConnection object, you can use any of the methods inherited from URLConnection. You can also use any of the several methods defined by HttpURLConnection. Here is a sampling:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static boolean getFollowRedirects()</td>
<td>Returns true if redirects are automatically followed and false otherwise. This feature is on by default.</td>
</tr>
<tr>
<td>String getRequestMethod()</td>
<td>Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available.</td>
</tr>
<tr>
<td>int getResponseCode() throws IOException</td>
<td>Returns the HTTP response code. −1 is returned if no response code can be obtained. An IOException is thrown if the connection fails.</td>
</tr>
<tr>
<td>String getResponseMessage() throws IOException</td>
<td>Returns the response message associated with the response code. Returns null if no message is available. An IOException is thrown if the connection fails.</td>
</tr>
<tr>
<td>static void setFollowRedirects(boolean how)</td>
<td>If how is true, then redirects are automatically followed. If how is false, redirects are not automatically followed. By default, redirects are automatically followed.</td>
</tr>
<tr>
<td>void setRequestMethod(String how) throws ProtocolException</td>
<td>Sets the method by which HTTP requests are made to that specified by how. The default method is GET, but other options, such as POST, are available. If how is invalid, a ProtocolException is thrown.</td>
</tr>
</tbody>
</table>

The following program demonstrates HttpURLConnection. It first establishes a connection to www.google.com. Then it displays the request method, the response code, and the response message. Finally, it displays the keys and values in the response header.
import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLConnectionDemo
{
    public static void main(String args[]) throws Exception {
        URL hp = new URL("http://www.google.com");

        HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

        // Display request method.
        System.out.println("Request method is " +
                          hpCon.getRequestMethod());

        // Display response code.
        System.out.println("Response code is " +
                          hpCon.getResponseCode());

        // Display response message.
        System.out.println("Response Message is " +
                          hpCon.getResponseMessage());

        // Get a list of the header fields and a set
        // of the header keys.
        Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
        Set<String> hdrField = hdrMap.keySet();

        System.out.println("\nHere is the header:");

        // Display all header keys and values.
        for(String k : hdrField) {
            System.out.println("Key: " + k +
                            " Value: " + hdrMap.get(k));
        }
    }
}

The output produced by the program is shown here. (Of course, the exact response returned by www.google.com will vary over time.)

Request method is GET
Response code is 200
Response Message is OK
Here is the header:
Key: null Value: [HTTP/1.1 200 OK]
Notice how the header keys and values are displayed. First, a map of the header keys and values is obtained by calling getHeaderFields() (which is inherited from URLConnection). Next, a set of the header keys is retrieved by calling keySet() on the map. Then the key set is cycled through by using a for-each style for loop. The value associated with each key is obtained by calling get() on the map.

The URI Class
The URI class encapsulates a Uniform Resource Identifier (URI). URIs are similar to URLs. In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource.

Cookies
The java.net package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session. The classes are CookieHandler, CookieManager, and HttpCookie. The interfaces are CookiePolicy and CookieStore. All but CookieHandler were added by Java SE 6. (CookieHandler was added by JDK 5.) The creation of a
CookieHandler was added by JDK 5. The creation of a stateful HTTP session is beyond the scope of this book.

NOTE

For information about using cookies with servlets, see Chapter 32.

TCP/IP Server Sockets
As mentioned earlier, Java has a different socket class that must be used for creating server applications. The ServerSocket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. ServerSockets are quite different from normal Sockets. When you create a ServerSocket, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an IOException under adverse conditions. Here are three of its constructors:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ServerSocket(int port)</code> throws IOException</td>
<td>Creates server socket on the specified port with a queue length of 50.</td>
</tr>
<tr>
<td><code>ServerSocket(int port, int maxQueue)</code> throws IOException</td>
<td>Creates a server socket on the specified port with a maximum queue length of <code>maxQueue</code>.</td>
</tr>
<tr>
<td><code>ServerSocket(int port, int maxQueue, InetAddress localAddress)</code> throws IOException</td>
<td>Creates a server socket on the specified port with a maximum queue length of <code>maxQueue</code>. On a multihomed host, <code>localAddress</code> specifies the IP address to which this socket binds.</td>
</tr>
</tbody>
</table>

ServerSocket has a method called `accept()`, which is a blocking call that will wait for a client to initiate communications and then return with a normal Socket.
Datagrams
TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the DatagramPacket object is the data container, while the DatagramSocket is the mechanism used to send or receive the DatagramPackets. Each is examined here.

DatagramSocket
DatagramSocket defines four public constructors. They are shown here:

DatagramSocket( ) throws SocketException
DatagramSocket(int port) throws SocketException

DatagramSocket(int port, InetAddress ipAddress) throws SocketException

DatagramSocket(SocketAddress address) throws SocketException

The first creates a DatagramSocket bound to any unused port on the local computer. The second creates a DatagramSocket bound to the port specified by port. The third constructs a DatagramSocket bound to the specified port and InetAddress. The fourth constructs a DatagramSocket bound to the specified SocketAddress. SocketAddress is an abstract class that is implemented by the concrete class InetSocketAddress. InetSocketAddress encapsulates an IP address with a port number. All can throw a SocketException if an error occurs while creating the socket.

DatagramSocket defines many methods. Two of the most important are send( ) and receive( ), which are shown here:

void send(DatagramPacket packet) throws IOException

void receive(DatagramPacket packet) throws IOException

The send( ) method sends a packet to the port specified by packet. The receive( ) method waits for a packet to be received from the port specified by packet and returns the result.

DatagramSocket also defines the close( ) method,
which closes the socket. Beginning with JDK 7, DatagramSocket implements AutoCloseable, which means that a DatagramSocket can be managed by a try-with-resources block.

Other methods give you access to various attributes associated with a DatagramSocket. Here is a sampling:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InetAddress getInetAddress()</td>
<td>If the socket is connected, then the address is returned. Otherwise, null is returned.</td>
</tr>
<tr>
<td>int getLocalPort( )</td>
<td>Returns the number of the local port.</td>
</tr>
<tr>
<td>int getPort( )</td>
<td>Returns the number of the port to which the socket is connected. It returns -1 if the socket is not connected to a port.</td>
</tr>
<tr>
<td>boolean isBound( )</td>
<td>Returns true if the socket is bound to an address. Returns false otherwise.</td>
</tr>
<tr>
<td>boolean isConnected( )</td>
<td>Returns true if the socket is connected to a server. Returns false otherwise.</td>
</tr>
<tr>
<td>void setSoTimeout(int millis) throws SocketException</td>
<td>Sets the time-out period to the number of milliseconds passed in millis.</td>
</tr>
</tbody>
</table>

DatagramPacket

DatagramPacket defines several constructors. Four are shown here:

DatagramPacket(byte data [], int size)
DatagramPacket(byte data [], int offset, int size)
DatagramPacket(byte data [], int size, InetAddress ipAddress, int port)
DatagramPacket(byte data [], int offset, int size, InetAddress ipAddress, int port)

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a DatagramSocket. The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a DatagramSocket to determine where
the data in the packet will be sent. The fourth form transmits packets beginning at the specified offset into the data. Think of the first two forms as building an “in box,” and the second two forms as stuffing and addressing an envelope.

DatagramPacket defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length. In general, the get methods are used on packets that are received and the set methods are used on packets that will be sent.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InetSocketAddress getAddress()</td>
<td>Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).</td>
</tr>
<tr>
<td>byte[] getData()</td>
<td>Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.</td>
</tr>
<tr>
<td>int getLength()</td>
<td>Returns the length of the valid data contained in the byte array that would be returned from the getData() method. This may not equal the length of the whole byte array.</td>
</tr>
<tr>
<td>int getOffset()</td>
<td>Returns the starting index of the data.</td>
</tr>
<tr>
<td>int getPort()</td>
<td>Returns the port number.</td>
</tr>
<tr>
<td>void setAddress(InetSocketAddress ipAddress)</td>
<td>Sets the address to which a packet will be sent. The address is specified by ipAddress.</td>
</tr>
<tr>
<td>void setData(byte[] data)</td>
<td>Sets the data to data, the offset to zero, and the length to number of bytes in data.</td>
</tr>
<tr>
<td>void setData(byte[] data, int idx, int size)</td>
<td>Sets the data to data, the offset to idx, and the length to size.</td>
</tr>
<tr>
<td>void setLength(int size)</td>
<td>Sets the length of the packet to size.</td>
</tr>
<tr>
<td>void setPort(int port)</td>
<td>Sets the port to port.</td>
</tr>
</tbody>
</table>

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are
/ Demonstrate datagrams.

import java.net.*;

class WriteServer {
    public static int serverPort = 998;
    public static int clientPort = 999;
    public static int buffer_size = 1024;
    public static DatagramSocket ds;
    public static byte buffer[] = new byte[buffer_size];

    public static void TheServer() throws Exception {
        int pos = 0;
        while (true) {
            int c = System.in.read();
            switch (c) {
                case -1:
                    System.out.println("Server Quits.");
                    ds.close();
                    return;
                case '\r':
                    break;
                case '\n':
                    ds.send(new DatagramPacket(buffer,pos,
                        InetAddress.getLoca
                    );
                    pos = 0;
                    break;
                default:
                    buffer[pos++] = (byte) c;
            }
        }
    }
}
public static void TheClient() throws Exception {
    while(true) {
        DatagramPacket p = new DatagramPacket(buffer, buffer.length);
        ds.receive(p);
        System.out.println(new String(p.getData(), 0, p.getLength()));
    }
}

public static void main(String args[]) throws Exception {
    if(args.length == 1) {
        ds = new DatagramSocket(serverPort);
        TheServer();
    } else {
        ds = new DatagramSocket(clientPort);
        TheClient();
    }
}

This sample program is restricted by the DatagramSocket constructor to running between two ports on the local machine. To use the program, run

    java WriteServer

in one window; this will be the client. Then run

    java WriteServer 1

This will be the server. Anything that is typed in the server window will be sent to the client window after a newline is received.
CHAPTER 22
The Applet Class

This chapter examines the Applet class, which provides the foundation for applets. The Applet class is contained in the java.applet package. Applet contains several methods that give you detailed control over the execution of your applet. In addition, java.applet also defines three interfaces: AppletContext, AudioClip, and AppletStub.

Two Types of Applets
It is important to state at the outset that there are two varieties of applets. The first are those based directly on the Applet class described in this chapter. These applets use the Abstract Window Toolkit (AWT) to provide the graphical user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class JApplet. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required. Thus, both AWT- and Swing-based applets are valid.

Because JApplet inherits Applet, all the features of
Applet are also available in JApplet, and most of the information in this chapter applies to both types of applets. Therefore, even if you are interested in only Swing applets, the information in this chapter is still relevant and necessary. Understand, however, that when creating Swing-based applets, some additional constraints apply and these are described later in this book, when Swing is covered.

**NOTE**

For information on building applets when using Swing, see [Chapter 30](#).

**Applet Basics**

*Chapter 13* introduced the general form of an applet and the steps necessary to compile and run one. Let’s begin by reviewing this information.

All applets are subclasses (either directly or indirectly) of Applet. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. The illustrations shown in this chapter were created with the standard applet viewer, called appletviewer, provided by the JDK. But you can use any applet viewer or browser you like.

Execution of an applet does not begin at `main()`. Actually, few applets even have `main()` methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be
explained shortly. Output to your applet’s window is not performed by System.out.println(). Rather, in non-Swing applets, output is handled with various AWT methods, such as drawString(), which outputs a string to a specified X,Y location. Input is also handled differently than in a console application. (Remember, Swing-based applets use the Swing classes to handle user interactions, and they are described later in this book.)

Before an applet can be used, a deployment strategy must be chosen. There are two basic approaches. The first is to use the Java Network Launch Protocol (JNLP). This approach offers the most flexibility, especially as it relates to rich Internet applications. For real-world applets that you create, JNLP will often be the best choice. However, a detailed discussion of JNLP is beyond the scope of this book. (See the JDK documentation for the latest details on JNLP.) Fortunately, JNLP is not required for the example applets shown here.

The second basic approach to deploying an applet is to specify the applet directly in an HTML file, without the use of JNLP. This is the original way that applets were launched when Java was created, and it is still used today—especially for simple applets. Furthermore, because of its inherent simplicity, it is the appropriate method for the applet examples described in this book. At the time of this writing, Oracle recommends the APPLET tag for this purpose. Therefore, the APPLET tag
is used in this book. (Be aware that the APPLET tag is currently deprecated by the HTML specification. The alternative is the OBJECT tag. You should check the JDK documentation in this regard for the latest recommendations.) When an APPLET tag is encountered in the HTML file, the specified applet will be executed by a Java-enabled web browser.

The use of the APPLET tag offers a secondary advantage when developing applets because it enables you to easily view and test the applet. To do so, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is an example of such a comment:

```html
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called MyApplet in a window that is 200 pixels wide and 60 pixels high. Because the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.
The Applet Class

The Applet class defines the methods shown in Table 22-1. Applet provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. Applet extends the AWT class Panel. In turn, Panel extends Container, which extends Component. These classes provide support for Java’s window-based, graphical interface. Thus, Applet provides all of the necessary support for window-based activities. (The AWT is described in detail in following chapters.)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void destroy( )</td>
<td>Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.</td>
</tr>
<tr>
<td>AccessibleContext getAccessibleContext( )</td>
<td>Returns the accessibility context for the invoking object.</td>
</tr>
<tr>
<td>AppletContext getAppletContext( )</td>
<td>Returns the context associated with the applet.</td>
</tr>
<tr>
<td>String getAppletInfo( )</td>
<td>Overrides of this method should return a string that describes the applet. The default implementation returns null.</td>
</tr>
<tr>
<td>AudioClip getAudioClip(URL url)</td>
<td>Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.</td>
</tr>
<tr>
<td>AudioClip getAudioClip(URL url, String clipName)</td>
<td>Returns an AudioClip object that encapsulates the audio clip found at the location specified by url and having the name specified by clipName.</td>
</tr>
<tr>
<td>URL getCodeBase( )</td>
<td>Returns the URL associated with the invoking applet.</td>
</tr>
<tr>
<td>URL getDocumentBase( )</td>
<td>Returns the URL of the HTML document that invokes the applet.</td>
</tr>
<tr>
<td>Image getImage(URL url)</td>
<td>Returns an Image object that encapsulates the image found at the location specified by url.</td>
</tr>
<tr>
<td>Image getImage(URL url, String imageName)</td>
<td>Returns an Image object that encapsulates the image found at the location specified by url and having the name specified by imageName.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>Locale getLocale()</code></td>
<td>Returns a <code>Locale</code> object that is used by various locale-sensitive classes and methods.</td>
</tr>
<tr>
<td><code>String getParameter(String paramName)</code></td>
<td>Returns the parameter associated with <code>paramName</code>. <code>null</code> is returned if the specified parameter is not found.</td>
</tr>
<tr>
<td><code>String[ ][ ] getParameterInfo()</code></td>
<td>Overrides of this method should return a <code>String</code> table that describes the parameters recognized by the applet. Each entry in the table</td>
</tr>
<tr>
<td></td>
<td>must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its</td>
</tr>
<tr>
<td></td>
<td>purpose. The default implementation returns <code>null</code>.</td>
</tr>
<tr>
<td><code>void init()</code></td>
<td>Called when an applet begins execution. It is the first method called for any applet.</td>
</tr>
<tr>
<td><code>boolean isActive()</code></td>
<td>Returns <code>true</code> if the applet has been started. It returns <code>false</code> if the applet has been stopped.</td>
</tr>
<tr>
<td><code>boolean isValidateRoot()</code></td>
<td>Returns <code>true</code>, which indicates that an applet is a validate root. (Added by JDK 7.)</td>
</tr>
<tr>
<td><code>static final AudioClip newAudioClip(URL url)</code></td>
<td>Returns an <code>AudioClip</code> object that encapsulates the audio clip found at the location specified by <code>url</code>. This method is similar to</td>
</tr>
<tr>
<td></td>
<td><code>getAudioClip()</code> except that it is static and can be executed without the need for an <code>Applet</code> object.</td>
</tr>
</tbody>
</table>
Applet Architecture

As a general rule, an applet is a GUI-based program. As such, its architecture is different from the console-based programs shown in the first part of this book. If you are already familiar with GUI programming, you will be right at home writing applets. If not, then there are a few key concepts you must understand.

First, applets are event driven. Although we won’t examine event handling until the following chapter, it is important to understand in a general way how the event-
driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs. The runtime system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point. For the most part, your applet should not enter a “mode” of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution. (You will see an example later in this chapter.)

Second, the user initiates interaction with an applet—not the other way around. As you know, in a console-based program, when the program needs input, it will prompt the user and then call some input method, such as readLine( ). This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet’s window, a mouse-clicked event is
generated. If the user presses a key while the applet’s window has input focus, a keypress event is generated. As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

While the architecture of an applet is not as easy to understand as that of a console-based program, Java makes it as simple as possible. If you have written programs for Windows (or other GUI-based operating systems), you know how intimidating that environment can be. Fortunately, Java provides a much cleaner approach that is more quickly mastered.

An Applet Skeleton
All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, init( ), start( ), stop( ), and destroy( ), apply to all applets and are defined by Applet. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them.

AWT-based applets (such as those discussed in this chapter) will also often override the paint( ) method, which is defined by the AWT Component class. This method is called when the applet’s output must be
redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

```java
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
    // Called first.
    public void init() {
        // initialization
    }

    // Called second, after init(). Also called whenever the applet is restarted. */
    public void start() {
        // start or resume execution
    }

    // Called when the applet is stopped.
    public void stop() {
        // suspends execution
    }

    // Called when applet is terminated. This is the last method executed. */
    public void destroy() {
        // perform shutdown activities
    }

    // Called when an applet’s window must be restored.
    public void paint(Graphics g) {
        // redisplay contents of window
    }
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following
Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

Let's look more closely at these methods.

init( )
The init( ) method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.
The \texttt{start()} method is called after \texttt{init()}. It is also called to restart an applet after it has been stopped. Whereas \texttt{init()} is called once—the first time an applet is loaded—\texttt{start()} is called each time an applet’s HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at \texttt{start()}.  

\texttt{paint()}  

The \texttt{paint()} method is called each time your applet’s output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. \texttt{paint()} is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, \texttt{paint()} is called. The \texttt{paint()} method has one parameter of type \texttt{Graphics}. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.  

\texttt{stop()}  

The \texttt{stop()} method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When \texttt{stop()} is called, the applet is probably running. You should use
stop( ) to suspend threads that don’t need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.

destroy( )
The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

Overriding update( )
In some situations, your applet may need to override another method defined by the AWT, called update( ). This method is called when your applet has requested that a portion of its window be redrawn. The default version of update( ) simply calls paint( ). However, you can override the update( ) method so that it performs more subtle repainting. In general, overriding update( ) is a specialized technique that is not applicable to all applets, and the examples in this chapter do not override update( ).

Simple Applet Display Methods
As we’ve mentioned, applets are displayed in a window, and AWT-based applets use the AWT to perform input and output. Although we will examine the methods, procedures, and techniques necessary to fully handle the AWT windowed environment in subsequent chapters, a
few are described here, because we will use them to write sample applets. (Remember, Swing-based applets are described later in this book.)

As described in Chapter 13, to output a string to an applet, use `drawString()`, which is a member of the `Graphics` class. Typically, it is called from within either `update()` or `paint()`. It has the following general form:

```java
void drawString(String message, int x, int y)
```

Here, `message` is the string to be output beginning at `x,y`. In a Java window, the upper-left corner is location 0,0. The `drawString()` method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin. (As you will see in later chapters, there are techniques that make this process easy.)

To set the background color of an applet’s window, use `setBackground()`. To set the foreground color (the color in which text is shown, for example), use `setForeground()`. These methods are defined by `Component`, and they have the following general forms:

```java
void setBackground(Color newColor)
void setForeground(Color newColor)
```

Here, `newColor` specifies the new color. The class `Color` defines the constants shown here that can be used to
specify colors:

<table>
<thead>
<tr>
<th>Color.black</th>
<th>Color.magenta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color.blue</td>
<td>Color.orange</td>
</tr>
<tr>
<td>Color.cyan</td>
<td>Color.pink</td>
</tr>
<tr>
<td>Color.darkGray</td>
<td>Color.red</td>
</tr>
<tr>
<td>Color.gray</td>
<td>Color.white</td>
</tr>
<tr>
<td>Color.green</td>
<td>Color.yellow</td>
</tr>
<tr>
<td>Color.lightGray</td>
<td></td>
</tr>
</tbody>
</table>

Uppercase versions of the constants are also defined.

The following example sets the background color to green and the text color to red:

```java
setBackground(Color.green);
setForeground(Color.red);
```

A good place to set the foreground and background colors is in the `init()` method. Of course, you can change these colors as often as necessary during the execution of your applet.

You can obtain the current settings for the background and foreground colors by calling `getBackground()` and `getForeground()`, respectively. They are also defined by `Component` and are shown here:

```java
Color getBackground()
Color getForeground()
```

Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the `init()`, `start()`, and `paint()` methods are called when an applet
This applet generates the window shown here:

![Applet Viewer: Sample](image)

The methods `stop()` and `destroy()` are not overridden,
because they are not needed by this simple applet.

Requesting Repainting
As a general rule, an applet writes to its window only when its `update()` or `paint()` method is called by the AWT. This raises an interesting question: How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? Remember, one of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the run-time system. It cannot create a loop inside `paint()` that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your applet’s window will be difficult at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls `repaint()`.

The `repaint()` method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet’s `update()` method, which, in its default implementation, calls `paint()`. Thus, for another part of your applet to output to its window, simply store the output and then call `repaint()`. The AWT will then execute a call to `paint()`, which can display the stored information. For example, if part of your applet needs to
output a string, it can store this string in a String variable and then call repaint(). Inside paint(), you will output the string using drawString().

The repaint() method has four forms. Let’s look at each one, in turn. The simplest version of repaint() is shown here:

```java
void repaint()
```

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

```java
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by left and top, and the width and height of the region are passed in width and height. These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling repaint() is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, update() might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that update() is only called sporadically.
This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of `repaint( )`:

```java
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
```

Here, `maxDelay` specifies the maximum number of milliseconds that can elapse before `update( )` is called. Beware, though. If the time elapses before `update( )` can be called, it isn’t called. There’s no return value or exception thrown, so you must be careful.

**NOTE**

It is possible for a method other than `paint( )` or `update( )` to output to an applet’s window. To do so, it must obtain a graphics context by calling `getGraphics( )` (defined by `Component`) and then use this context to output to the window. However, for most applications, it is better and easier to route window output through `paint( )` and to call `repaint( )` when the contents of the window change.

**A Simple Banner Applet**

To demonstrate `repaint( )`, a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet’s window. Since the scrolling of
The message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. The banner applet is shown here:

/* A simple banner applet.

This applet creates a thread that scrolls the message contained in msg right to left across the applet's window.
*/
import java.awt.*;
import java.applet.*/

<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable {
    String msg = "A Simple Moving Banner.";
    Thread t = null;
    int state;
    volatile boolean stopFlag;

    // Set colors and initialize thread.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
    }

    // Start thread
    public void start() {
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }
public void run() { 

    // Redisplay banner
    for(; ; ) { 
        try {
            repaint();
            Thread.sleep(250);
            if(stopFlag)
                break;
        } catch(InterruptedException e) {} 
    }
}

// Pause the banner.
public void stop() {
    stopFlag = true;
    t = null;
}

// Display the banner.
public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
}

Following is sample output:

![Applet Viewer: SimpleBanner](image)

Applet

ple Moving Banner. A Sim

Applet started.

Let's take a close look at how this applet operates. First, notice that SimpleBanner extends Applet, as
expected, but it also implements Runnable. This is necessary, since the applet will be creating a second thread of execution that will be used to scroll the banner. Inside init( ), the foreground and background colors of the applet are set.

After initialization, the run-time system calls start( ) to start the applet running. Inside start( ), a new thread of execution is created and assigned to the Thread variable t. Then, the boolean variable stopFlag, which controls the execution of the applet, is set to false. Next, the thread is started by a call to t.start( ). Remember that t.start( ) calls a method defined by Thread, which causes run( ) to begin executing. It does not cause a call to the version of start( ) defined by Applet. These are two separate methods.

Inside run( ), a call to repaint( ) is made. This eventually causes the paint( ) method to be called, and the rotated contents of msg are displayed. Between each iteration, run( ) sleeps for a quarter of a second. The net effect is that the contents of msg are scrolled right to left in a constantly moving display. The stopFlag variable is checked on each iteration. When it is true, the run( ) method terminates.

If a browser is displaying the applet when a new page is viewed, the stop( ) method is called, which sets stopFlag to true, causing run( ) to terminate. This is the mechanism used to stop the thread when its page is no
longer in view. When the applet is brought back into view, start() is once again called, which starts a new thread to execute the banner.

Using the Status Window
In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call showStatus() with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

The following applet demonstrates showStatus():

```java
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/

class StatusWindow extends Applet {
    public void init() {
        setBackground(Color.cyan);
    }

    // Display msg in applet window.
    public void paint(Graphics g) {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
    }
}
```
Sample output from this program is shown here:

![Applet Viewer: StatusWindow](image)

Applet

This is in the applet window.

This is shown in the status window.

The HTML APPLET Tag

As mentioned earlier, at the time of this writing, Oracle recommends that the APPLET tag be used to manually start an applet when JNLP is not used. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. So far, we have been using only a simplified form of the APPLET tag. Now it is time to take a closer look at it.

The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional.

```html
<APPLET
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = alternateText]
  [NAME = appletInstanceName]
  WIDTH = pixels HEIGHT = pixels
  [ALIGN = alignment]
  [VSPACE = pixels] [HSPACE = pixels]
</APPLET>
```
Let's take a look at each part now.

**CODEBASE**

CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet’s executable class file (specified by the CODE tag). The HTML document’s URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

**CODE**

CODE is a required attribute that gives the name of the file containing your applet’s compiled .class file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

**ALT**

The ALT tag is an optional attribute used to specify a
The **ALT tag** is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the **APPLET** tag but can’t currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don’t support applets.

**NAME**

**NAME** is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use `getApplet()`, which is defined by the AppletContext interface.

**WIDTH and HEIGHT**

**WIDTH** and **HEIGHT** are required attributes that give the size (in pixels) of the applet display area.

**ALIGN**

**ALIGN** is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSTEMIDDLE, and ABSBOTTOM.

**VSPACE and HSPACE**

These attributes are optional. **VSPACE** specifies the space, in pixels, above and below the applet. **HSPACE**
specifies the space, in pixels, on each side of the applet. They’re treated the same as the IMG tag’s VSPACE and HSPACE attributes.

PARAM NAME and VALUE
The PARAM tag allows you to specify applet-specific arguments. Applets access their attributes with the getParameter() method.

Other valid APPLET attributes include ARCHIVE, which lets you specify one or more archive files, and OBJECT, which specifies a saved version of the applet. In general, an APPLET tag should include only a CODE or an OBJECT attribute, but not both.

Passing Parameters to Applets
As just discussed, the APPLET tag allows you to pass parameters to your applet. To retrieve a parameter, use the getParameter() method. It returns the value of the specified parameter in the form of a String object. Thus, for numeric and boolean values, you will need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:
import java.awt.*;
import java.applet.*;

/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet {
    String fontName;
    int fontSize;
    float leading;
    boolean active;

    // Initialize the string to be displayed.
    public void start() {
        String param;

        fontName = getParameter("fontName");
        if(fontName == null)
            fontName = "Not Found";

        param = getParameter("fontSize");
        try {
            if(param != null)
                fontSize = Integer.parseInt(param);
            else
                fontSize = 0;
        } catch(NumberFormatException e) {
            fontSize = -1;
        }
    }
}
Sample output from this program is shown here:

As the program shows, you should test the return values from `getParameter()` function. If a parameter isn’t available, `getParameter()` will return null. Also, conversions to numeric types must be attempted in a try statement that catches `NumberFormatException`. Uncaught exceptions should never occur within an
Improving the Banner Applet

It is possible to use a parameter to enhance the banner applet shown earlier. In the previous version, the message being scrolled was hard-coded into the applet. However, passing the message as a parameter allows the banner applet to display a different message each time it is executed. This improved version is shown here. Notice that the APPLET tag at the top of the file now specifies a parameter called message that is linked to a quoted string.
// A parameterized banner
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java makes the Web move!">
</applet>
public class ParamBanner extends Applet implements Runnable {
    String msg;
    Thread t = null;
    int state;
    volatile boolean stopFlag;

    // Set colors and initialize thread.
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
    }

    // Start thread
    public void start() {
        msg = getParameter("message");
        if(msg == null) msg = "Message not found.";
        msg = " " + msg;
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }
}
Often, you will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet (the document base) and the directory from which the applet’s class file was loaded (the code base). These directories are returned as URL objects (described in Chapter 20) by `getDocumentBase()` and `getCodeBase()`.
getCodeBase(). They can be concatenated with a string that names the file you want to load. To actually load another file, you will use the showDocument( ) method defined by the AppletContext interface, discussed in the next section.

The following applet illustrates these methods:

```java
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
 <applet code="Bases" width=300 height=50>
 </applet>
 */

public class Bases extends Applet {
    // Display code and document bases.
    public void paint(Graphics g) {
        String msg;

        URL url = getCodeBase(); // get code base
        msg = "Code base: " + url.toString();
        g.drawString(msg, 10, 20);

        url = getDocumentBase(); // get document base
        msg = "Document base: " + url.toString();
        g.drawString(msg, 10, 40);
    }
}
```

Sample output from this program is shown here:
AppletContext and showDocument()  
One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than simple text-based links. To allow your applet to transfer control to another URL, you must use the showDocument( ) method defined by the AppletContext interface. AppletContext is an interface that lets you get information from the applet’s execution environment. The methods defined by AppletContext are shown in Table 22-2. The context of the currently executing applet is obtained by a call to the getAppletContext( ) method defined by Applet.

Within an applet, once you have obtained the applet’s context, you can bring another document into view by calling showDocument( ). This method has no return value and throws no exception if it fails, so use it carefully. There are two showDocument( ) methods. The method showDocument(URL) displays the document at the specified URL. The method showDocument(URL, String) displays the specified document at the specified location within the browser window. Valid arguments for where are “_self” (show in current frame), “_parent” (show in parent frame), “_top” (show in topmost frame), and “_blank” (show in new browser window). You can also specify a name, which causes the document to be shown in a new browser window by that name.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applet getApplet(String appletName)</td>
<td>Returns the applet specified by <em>appletName</em> if it is within the current applet context. Otherwise, null is returned.</td>
</tr>
<tr>
<td>Enumeration&lt;Applet&gt; getApplets( )</td>
<td>Returns an enumeration that contains all of the applets within the current applet context.</td>
</tr>
<tr>
<td>AudioClip getAudioClip(URL url)</td>
<td>Returns an AudioClip object that encapsulates the audio clip found at the location specified by <em>url</em>.</td>
</tr>
<tr>
<td>Image getImage(URL url)</td>
<td>Returns an Image object that encapsulates the image found at the location specified by <em>url</em>.</td>
</tr>
<tr>
<td>InputStream getStream(String key)</td>
<td>Returns the stream linked to <em>key</em>. Keys are linked to streams by using the setStream() method. A null reference is returned if no stream is linked to <em>key</em>.</td>
</tr>
<tr>
<td>Iterator&lt;String&gt; getStreamKeys( )</td>
<td>Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See getStream() and setStream().</td>
</tr>
<tr>
<td>void setStream(String key, InputStream strm)</td>
<td>Links the stream specified by <em>strm</em> to the key passed in <em>key</em>. The <em>key</em> is deleted from the invoking object if <em>strm</em> is null.</td>
</tr>
<tr>
<td>void showDocument(URL url)</td>
<td>Brings the document at the URL specified by <em>url</em> into view. This method may not be supported by applet viewers.</td>
</tr>
<tr>
<td>void showDocument(URL url, String where)</td>
<td>Brings the document at the URL specified by <em>url</em> into view. This method may not be supported by applet viewers. The placement of the document is specified by <em>where</em> as described in the text.</td>
</tr>
<tr>
<td>void showStatus(String str)</td>
<td>Displays <em>str</em> in the status window.</td>
</tr>
</tbody>
</table>

Table 22-2 The Methods Defined by the AppletContext Interface

The following applet demonstrates AppletContext and showDocument(). Upon execution, it obtains the current applet context and uses that context to transfer control to a file called Test.html. This file must be in the same directory as the applet. Test.html can contain any valid hypertext that you like.
The AudioClip Interface
The AudioClip interface defines these methods: `play()` (play a clip from the beginning), `stop()` (stop playing the clip), and `loop()` (play the loop continuously). After you have loaded an audio clip using `getAudioClip()`, you can use these methods to play it.

The AppletStub Interface
The AppletStub interface provides the means by which an applet and the browser (or applet viewer) communicate. Your code will not typically implement this interface.
Outputting to the Console

Although output to an applet’s window must be accomplished through GUI-based methods, such as drawString( ), it is still possible to use console output in your applet—especially for debugging purposes. In an applet, when you call a method such as System.out.println( ), the output is not sent to your applet’s window. Instead, it appears either in the console session in which you launched the applet viewer or in the Java console that is available in some browsers. Use of console output for purposes other than debugging is discouraged, since it violates the design principles of the graphical interface most users will expect.
This chapter examines an important aspect of Java: the event. Event handling is fundamental to Java programming because it is integral to the creation of applets and other types of GUI-based programs. As explained in Chapter 22, applets are event-driven programs that use a graphical user interface to interact with the user. Furthermore, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Thus, you cannot write these types of programs without a solid command of event handling. Events are supported by a number of packages, including java.util, java.awt, and java.awt.event.

Most events to which your program will respond are generated when the user interacts with a GUI-based program. These are the types of events examined in this chapter. They are passed to your program in a variety of ways, with the specific method dependent upon the actual event. There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.

This chapter begins with an overview of Java’s event handling mechanism. It then examines the main event classes and interfaces used by the AWT and develops
several examples that demonstrate the fundamentals of event processing. This chapter also explains how to use adapter classes, inner classes, and anonymous inner classes to streamline event handling code. The examples provided in the remainder of this book make frequent use of these techniques.

NOTE

This chapter focuses on events related to GUI-based programs. However, events are also occasionally used for purposes not directly related to GUI-based programs. In all cases, the same basic event handling techniques apply.

Two Event Handling Mechanisms

Before beginning our discussion of event handling, an important point must be made: The way in which events are handled changed significantly between the original version of Java (1.0) and modern versions of Java, beginning with version 1.1. The 1.0 method of event handling is still supported, but it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs in this book.

The Delegation Event Model

The modern approach to handling events is based on the
delegation event model, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

NOTE
Java also allows you to process events without using the delegation event model. This can be done by extending an AWT component. This technique is
discussed at the end of Chapter 25. However, the delegation event model is the preferred design for the reasons just cited.

The following sections define events and describe the roles of sources and listeners.

Events
In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources
A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the
listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

```java
public void add Type Listener (Type Listener el)
```

Here, Type is the name of the event, and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called `addKeyListener()`. The method that registers a mouse motion listener is called `addMouseMotionListener()`. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

```java
public void add Type Listener (Type Listener el)
    throws java.util.TooManyListenersException
```

Here, Type is the name of the event, and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:
public void remove Type Listener (Type Listener el)

Here, Type is the name of the event, and el is a reference to the event listener. For example, to remove a keyboard listener, you would call removeKeyListener().

The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners
A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Many other listener interfaces are discussed later in this and other chapters.

Event Classes
The classes that represent events are at the core of Java’s
event handling mechanism. Thus, a discussion of event handling must begin with the event classes. It is important to understand, however, that Java defines several types of events and that not all event classes can be discussed in this chapter. The most widely used events are those defined by the AWT and those defined by Swing. This chapter focuses on the AWT events. (Most of these events also apply to Swing.) Several Swing-specific events are described in Chapter 30, when Swing is covered.

At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events. Its one constructor is shown here:

```
EventObject(Object src)
```

Here, src is the object that generates this event.

EventObject contains two methods: getSource( ) and toString( ). The getSource( ) method returns the source of the event. Its general form is shown here:

```
Object getSource( )
```

As expected, toString( ) returns the string equivalent of the event.

The class AWTEvent, defined within the java.awt package, is a subclass of EventObject. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. ItsgetID( ) method
can be used to determine the type of the event. The signature of this method is shown here:

    int getID( )

Additional details about AWTEvent are provided at the end of Chapter 25. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of AWTEvent.

To summarize:

• EventObject is a superclass of all events.

• AWTEvent is a superclass of all AWT events that are handled by the delegation event model.

The package java.awt.event defines many types of events that are generated by various user interface elements. Table 23-1 shows several commonly used event classes and provides a brief description of when they are generated. Commonly used constructors and methods in each class are described in the following sections.
Table 23-1 Commonly Used Event Classes in java.awt.event

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.</td>
</tr>
<tr>
<td>AdjustmentEvent</td>
<td>Generated when a scroll bar is manipulated.</td>
</tr>
<tr>
<td>ComponentEvent</td>
<td>Generated when a component is hidden, moved, resized, or becomes visible.</td>
</tr>
<tr>
<td>ContainerEvent</td>
<td>Generated when a component is added to or removed from a container.</td>
</tr>
<tr>
<td>FocusEvent</td>
<td>Generated when a component gains or loses keyboard focus.</td>
</tr>
<tr>
<td>InputEvent</td>
<td>Abstract superclass for all component input event classes.</td>
</tr>
<tr>
<td>ItemEvent</td>
<td>Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.</td>
</tr>
<tr>
<td>KeyEvent</td>
<td>Generated when input is received from the keyboard.</td>
</tr>
<tr>
<td>MouseEvent</td>
<td>Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.</td>
</tr>
<tr>
<td>MouseWheelEvent</td>
<td>Generated when the mouse wheel is moved.</td>
</tr>
<tr>
<td>TextEvent</td>
<td>Generated when the value of a text area or text field is changed.</td>
</tr>
<tr>
<td>WindowEvent</td>
<td>Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.</td>
</tr>
</tbody>
</table>

The ActionEvent Class

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK, and SHIFTMASK. In addition, there is an integer constant, ACTION_PERFORMED, which can be used to identify action events.

ActionEvent has these three constructors:

ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)

ActionEvent(Object src, int type, String cmd, long when, int modifiers)

Here, src is a reference to the object that generated this event. The type of the event is specified by type, and its command string is cmd. The argument modifiers indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The when parameter specifies when the event occurred.

You can obtain the command name for the invoking ActionEvent object by using the getActionCommand() method, shown here:

String getActionCommand() 

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The getModifiers() method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

int getModifiers() 

The method getWhen() returns the time at which the event took place. This is called the event’s timestamp.
The getWhen( ) method is shown here:

```java
long getWhen()
```

The AdjustmentEvent Class

An AdjustmentEvent is generated by a scroll bar. There are five types of adjustment events. The AdjustmentEvent class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK_DECREMENT</td>
<td>The user clicked inside the scroll bar to decrease its value.</td>
</tr>
<tr>
<td>BLOCK_INCREMENT</td>
<td>The user clicked inside the scroll bar to increase its value.</td>
</tr>
<tr>
<td>TRACK</td>
<td>The slider was dragged.</td>
</tr>
<tr>
<td>UNIT_DECREMENT</td>
<td>The button at the end of the scroll bar was clicked to decrease its value.</td>
</tr>
<tr>
<td>UNIT_INCREMENT</td>
<td>The button at the end of the scroll bar was clicked to increase its value.</td>
</tr>
</tbody>
</table>

In addition, there is an integer constant, ADJUSTMENT_VALUE_CHANGED, that indicates that a change has occurred.

Here is one AdjustmentEvent constructor:

```java
AdjustmentEvent(Adjustable src, int id, int type, int data)
```

Here, src is a reference to the object that generated this event. The id specifies the event. The type of the adjustment is specified by type, and its associated data is data.

The getAdjustable( ) method returns the object that generated the event. Its form is shown here:
Adjustable getAdjustable()

The type of the adjustment event may be obtained by the getAdjustmentType() method. It returns one of the constants defined by AdjustmentEvent. The general form is shown here:

    int getAdjustmentType()

The amount of the adjustment can be obtained from the getValue() method, shown here:

    int getValue()

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

The ComponentEvent Class

A ComponentEvent is generated when the size, position, or visibility of a component is changed. There are four types of component events. The ComponentEvent class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

<table>
<thead>
<tr>
<th>COMPONENT_HIDDEN</th>
<th>The component was hidden.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPONENT_MOVED</td>
<td>The component was moved.</td>
</tr>
<tr>
<td>COMPONENT_RESIZED</td>
<td>The component was resized.</td>
</tr>
<tr>
<td>COMPONENT_SHOWN</td>
<td>The component became visible.</td>
</tr>
</tbody>
</table>

ComponentEvent has this constructor:

    ComponentEvent(Component src, int type)
Here, src is a reference to the object that generated this event. The type of the event is specified by type.

ComponentEvent is the superclass either directly or indirectly of ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent, among others.

The getComponent() method returns the component that generated the event. It is shown here:

Component getComponent() 

The ContainerEvent Class

A ContainerEvent is generated when a component is added to or removed from a container. There are two types of container events. The ContainerEvent class defines int constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED. They indicate that a component has been added to or removed from the container.

ContainerEvent is a subclass of ComponentEvent and has this constructor:

ContainerEvent( Component src,  int type,  Component comp) 

Here, src is a reference to the container that generated this event. The type of the event is specified by type, and the component that has been added to or removed from the container is comp.
You can obtain a reference to the container that generated this event by using the `getContainer()` method, shown here:

```java
Container getContainer()
```

The `getChild()` method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

```java
Component getChild()
```

The FocusEvent Class

A FocusEvent is generated when a component gains or loses input focus. These events are identified by the integer constants `FOCUS_GAINED` and `FOCUS_LOST`.

FocusEvent is a subclass of ComponentEvent and has these constructors:

```java
FocusEvent(Component src, int type)
FocusEvent(Component src, int type, boolean temporaryFlag)
FocusEvent(Component src, int type, boolean temporaryFlag, Component other)
```

Here, `src` is a reference to the component that generated this event. The type of the event is specified by `type`. The argument `temporaryFlag` is set to `true` if the focus event is temporary. Otherwise, it is set to `false`. (A temporary focus event occurs as a result of another user interface...
operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)

The other component involved in the focus change, called the opposite component, is passed in other. Therefore, if a FOCUS_GAINED event occurred, other will refer to the component that lost focus. Conversely, if a FOCUS_LOST event occurred, other will refer to the component that gains focus.

You can determine the other component by calling getOppositeComponent( ), shown here:

Component getOppositeComponent( )

The opposite component is returned.

The isTemporary( ) method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary( )

The method returns true if the change is temporary. Otherwise, it returns false.

The InputEvent Class
The abstract class InputEvent is a subclass of ComponentEvent and is the superclass for component input events. Its subclasses are KeyEvent and MouseEvent.

InputEvent defines several integer constants that
represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the `InputEvent` class defined the following eight values to represent the modifiers:

```
+----------------+----------------+----------------+
|    ALT_MASK    |    BUTTON2_MASK|      META_MASK |
|  ALT_GRAPH_MASK|    BUTTON3_MASK|      SHIFT_MASK|
|  BUTTON1_MASK  |       CTRL_MASK|              |
```

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

```
+----------------+----------------+----------------+
|     ALT_DOWN_M |    BUTTON2_DOWN_M|      META_DOWN_M |
|  ALT_GRAPH_DOWN_M|    BUTTON3_DOWN_M|      SHIFT_DOWN_M |
|  BUTTON1_DOWN_M |       CTRL_DOWN_M|              |
```

When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers.

To test if a modifier was pressed at the time an event is generated, use the `isAltDown()`, `isAltGraphDown()`, `isControlDown()`, `isMetaDown()`, and `isShiftDown()` methods. The forms of these methods are shown here:

```java
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```
You can obtain a value that contains all of the original modifier flags by calling the `getModifiers()` method. It is shown here:

```java
int getModifiers()
```

You can obtain the extended modifiers by calling `getModifiersEx()`, which is shown here:

```java
int getModifiersEx()
```

### The ItemEvent Class

An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this book.) There are two types of item events, which are identified by the following integer constants:

<table>
<thead>
<tr>
<th>Delected</th>
<th>The user deselected an item.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTED</td>
<td>The user selected an item.</td>
</tr>
</tbody>
</table>

In addition, `ItemEvent` defines one integer constant, `ITEM_STATE_CHANGED`, that signifies a change of state.

`ItemEvent` has this constructor:

```java
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

Here, `src` is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by `type`. The
specific item that generated the item event is passed in entry. The current state of that item is in state.

The getItem() method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

Object getItem()

The getItemSelectable() method can be used to obtain a reference to the ItemSelectable object that generated an event. Its general form is shown here:

ItemSelectable getItemSelectable()

Lists and choices are examples of user interface elements that implement the ItemSelectable interface.

The getStateChange() method returns the state change (that is, SELECTED or DESELECTED) for the event. It is shown here:

int getStateChange()

The KeyEvent Class
A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RECEIVED, and KEY_TYPED. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For
example, pressing `SHIFT` does not generate a character.

There are many other integer constants that are defined by `KeyEvent`. For example, `VK_0` through `VK_9` and `VK_A` through `VK_Z` define the ASCII equivalents of the numbers and letters. Here are some others:

<table>
<thead>
<tr>
<th>VK_ALT</th>
<th>VK_DOWN</th>
<th>VK_LEFT</th>
<th>VK_RIGHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_CANCEL</td>
<td>VK_ENTER</td>
<td>VK_PAGE_DOWN</td>
<td>VK_SHIFT</td>
</tr>
<tr>
<td>VK_CONTROL</td>
<td>VK_ESCAPE</td>
<td>VK_PAGE_UP</td>
<td>VK_UP</td>
</tr>
</tbody>
</table>

The `VK` constants specify virtual key codes and are independent of any modifiers, such as control, shift, or `alt`.

`KeyEvent` is a subclass of `InputEvent`. Here is one of its constructors:

```java
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
```

Here, `src` is a reference to the component that generated the event. The type of the event is specified by `type`. The system time at which the key was pressed is passed in `when`. The `modifiers` argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as `VK_UP`, `VK_A`, and so forth, is passed in `code`. The character equivalent (if one exists) is passed in `ch`. If no valid character exists, then `ch` contains `CHAR_UNDEFINED`. For `KEY_TYPED` events, `code` will contain `VK_UNDEFINED`.

The `KeyEvent` class defines several methods, but
probably the most commonly used ones are `getKeyChar()` which returns the character that was entered, and `getKeyCode()` which returns the key code. Their general forms are shown here:

```
char getKeyChar()
int getKeyCode()
```

If no valid character is available, then `getKeyChar()` returns `CHAR_UNDEFINED`. When a `KEY_TYPED` event occurs, `getKeyCode()` returns `VK_UNDEFINED`.

**The MouseEvent Class**

There are eight types of mouse events. The `MouseEvent` class defines the following integer constants that can be used to identify them:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOUSE_CLICKED</td>
<td>The user clicked the mouse.</td>
</tr>
<tr>
<td>MOUSE_DRAGGED</td>
<td>The user dragged the mouse.</td>
</tr>
<tr>
<td>MOUSE_ENTERED</td>
<td>The mouse entered a component.</td>
</tr>
<tr>
<td>MOUSE_EXITED</td>
<td>The mouse exited from a component.</td>
</tr>
<tr>
<td>MOUSE_MOVED</td>
<td>The mouse moved.</td>
</tr>
<tr>
<td>MOUSE_PRESSED</td>
<td>The mouse was pressed.</td>
</tr>
<tr>
<td>MOUSE_RELEASED</td>
<td>The mouse was released.</td>
</tr>
<tr>
<td>MOUSE_WHEEL</td>
<td>The mouse wheel was moved.</td>
</tr>
</tbody>
</table>

`MouseEvent` is a subclass of `InputEvent`. Here is one of its constructors:

```java
MouseEvent(Component src, int type, long when, int modifiers,
            int x, int y, int clicks, boolean
```
Here, src is a reference to the component that generated the event. The type of the event is specified by type. The system time at which the mouse event occurred is passed in when. The modifiers argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in x and y. The click count is passed in clicks. The triggersPopup flag indicates if this event causes a pop-up menu to appear on this platform.

Two commonly used methods in this class are getX( ) and getY( ). These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

```java
int getX()
int getY()
```

Alternatively, you can use the getPoint( ) method to obtain the coordinates of the mouse. It is shown here:

```java
Point getPoint()
```

It returns a Point object that contains the X,Y coordinates in its integer members: x and y.

The translatePoint( ) method changes the location of the event. Its form is shown here:

```java
void translatePoint(int x, int y)
```
Here, the arguments x and y are added to the coordinates of the event.

The `getClickCount()` method obtains the number of mouse clicks for this event. Its signature is shown here:

```java
int getClickCount()
```

The `isPopupTrigger()` method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

```java
boolean isPopupTrigger()
```

Also available is the `getButton()` method, shown here:

```java
int getButton()
```

It returns a value that represents the button that caused the event. The return value will be one of these constants defined by `MouseEvent`:

<table>
<thead>
<tr>
<th>NOBUTTON</th>
<th>BUTTON1</th>
<th>BUTTON2</th>
<th>BUTTON3</th>
</tr>
</thead>
</table>

The `NOBUTTON` value indicates that no button was pressed or released.

Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

```java
Point getLocationOnScreen()
int getXOnScreen()
```
int getYOnScreen()

The getLocationOnScreen() method returns a Point object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The MouseWheelEvent Class

The MouseWheelEvent class encapsulates a mouse wheel event. It is a subclass of MouseEvent. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. MouseWheelEvent defines these two integer constants:

<table>
<thead>
<tr>
<th>WHEEL_BLOCK_SCROLL</th>
<th>A page-up or page-down scroll event occurred.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHEEL_UNIT_SCROLL</td>
<td>A line-up or line-down scroll event occurred.</td>
</tr>
</tbody>
</table>

Here is one of the constructors defined by MouseWheelEvent:

```
MouseWheelEvent(Component src, int type, long when, int modifiers,
                 int x, int y, int clicks, boolean triggersPopup,
                 int scrollHow, int amount, int count)
```

Here, src is a reference to the object that generated the event. The type of the event is specified by type. The system time at which the mouse event occurred is passed in when. The modifiers argument indicates which
Modifiers were pressed when the event occurred. The coordinates of the mouse are passed in x and y. The number of clicks is passed in clicks. The triggersPopup flag indicates if this event causes a pop-up menu to appear on this platform. The scrollHow value must be either `WHEEL_UNIT_SCROLL` or `WHEEL_BLOCK_SCROLL`. The number of units to scroll is passed in amount. The count parameter indicates the number of rotational units that the wheel moved.

`MouseWheelEvent` defines methods that give you access to the wheel event. To obtain the number of rotational units, call `getWheelRotation()`, shown here:

```java
int getWheelRotation()
```

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. JDK 7 adds a method called `getPreciseWheelRotation()`, which supports high-resolution wheels. It works like `getWheelRotation()`, but returns a double.

To obtain the type of scroll, call `getScrollType()`, shown next:

```java
int getScrollType()
```

It returns either `WHEEL_UNIT_SCROLL` or `WHEEL_BLOCK_SCROLL`.

If the scroll type is `WHEEL_UNIT_SCROLL`, you can
obtain the number of units to scroll by calling getScrollAmount(). It is shown here:

```java
int getScrollAmount()
```

The TextEvent Class

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. TextEvent defines the integer constant TEXT_VALUE_CHANGED.

The one constructor for this class is shown here:

```java
TextEvent(Object src, int type)
```

Here, src is a reference to the object that generated this event. The type of the event is specified by type.

The TextEvent object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the TextEvent class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class

There are ten types of window events. The WindowEvent class defines integer constants that can be used to identify
WindowEvent is a subclass of ComponentEvent. It defines several constructors. The first is

WindowEvent(Window src, int type)

Here, src is a reference to the object that generated this event. The type of the event is type.

The next three constructors offer more detailed control:

WindowEvent(Window src, int type, Window other)
WindowEvent(Window src, int type, int fromState, int toState)
WindowEvent(Window src, int type, Window other, int fromState, int toState)

Here, other specifies the opposite window when a focus or activation event occurs. The fromState specifies the prior state of the window, and toState specifies the new state that the window will have when a window state
change occurs.

A commonly used method in this class is `getWindow()`. It returns the Window object that generated the event. Its general form is shown here:

```java
Window getWindow()
```

WindowEvent also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

```java
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Sources of Events

Table 23-2 lists some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, any class derived from `Component`, such as `Applet`, can generate events. For example, you can receive key and mouse events from an applet. (You may also build your own components that generate events.) In this chapter, we will be handling only mouse and keyboard events, but the following two chapters will be handling events from the sources shown in Table 23-2.
As explained, the delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the `java.awt.event` package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 23-3 lists commonly used listener interfaces and provides a brief description of the methods that they define. The following sections examine the specific methods that are contained in each interface.

**Table 23-2 Event Source Examples**

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Generates action events when the button is pressed.</td>
</tr>
<tr>
<td>Check box</td>
<td>Generates item events when the check box is selected or deselected.</td>
</tr>
<tr>
<td>Choice</td>
<td>Generates item events when the choice is changed.</td>
</tr>
<tr>
<td>List</td>
<td>Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.</td>
</tr>
<tr>
<td>Menu item</td>
<td>Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.</td>
</tr>
<tr>
<td>Scroll bar</td>
<td>Generates adjustment events when the scroll bar is manipulated.</td>
</tr>
<tr>
<td>Text components</td>
<td>Generates text events when the user enters a character.</td>
</tr>
<tr>
<td>Window</td>
<td>Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.</td>
</tr>
</tbody>
</table>
Table 23-3 Commonly Used Event Listener Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionListener</td>
<td>Defines one method to receive action events.</td>
</tr>
<tr>
<td>AdjustmentListener</td>
<td>Defines one method to receive adjustment events.</td>
</tr>
<tr>
<td>ComponentListener</td>
<td>Defines four methods to recognize when a component is hidden, moved, resized, or shown.</td>
</tr>
<tr>
<td>ContainerListener</td>
<td>Defines two methods to recognize when a component is added to or removed from a container.</td>
</tr>
<tr>
<td>FocusListener</td>
<td>Defines two methods to recognize when a component gains or loses keyboard focus.</td>
</tr>
<tr>
<td>ItemListener</td>
<td>Defines one method to recognize when the state of an item changes.</td>
</tr>
<tr>
<td>KeyListener</td>
<td>Defines three methods to recognize when a key is pressed, released, or typed.</td>
</tr>
<tr>
<td>MouseListener</td>
<td>Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.</td>
</tr>
<tr>
<td>MouseMotionListener</td>
<td>Defines two methods to recognize when the mouse is dragged or moved.</td>
</tr>
<tr>
<td>MouseWheelListener</td>
<td>Defines one method to recognize when the mouse wheel is moved.</td>
</tr>
<tr>
<td>TextListener</td>
<td>Defines one method to recognize when a text value changes.</td>
</tr>
<tr>
<td>WindowFocusListener</td>
<td>Defines two methods to recognize when a window gains or loses input focus.</td>
</tr>
<tr>
<td>WindowListener</td>
<td>Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.</td>
</tr>
</tbody>
</table>

The ActionListener Interface

This interface defines the actionPerformed( ) method that is invoked when an action event occurs. Its general form is shown here:

```java
void actionPerformed(ActionEvent ae)
```

The AdjustmentListener Interface

This interface defines the adjustmentValueChanged( ) method that is invoked when an adjustment event occurs. Its general form is shown here:
void adjustmentValueChanged(AdjustmentEvent ae)

The ComponentListener Interface
This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)

The ContainerListener Interface
This interface contains two methods. When a component is added to a container, componentAdded( ) is invoked. When a component is removed from a container, componentRemoved( ) is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)

The FocusListener Interface
This interface defines two methods. When a component obtains keyboard focus, focusGained( ) is invoked. When a component loses keyboard focus, focusLost( ) is called. Their general forms are shown here:

void focusGained(FocusEvent fe)
The ItemListener Interface
This interface defines the `itemStateChanged()` method that is invoked when the state of an item changes. Its general form is shown here:

```java
void itemStateChanged(ItemEvent ie)
```

The KeyListener Interface
This interface defines three methods. The `keyPressed()` and `keyReleased()` methods are invoked when a key is pressed and released, respectively. The `keyTyped()` method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

```java
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

The MouseListener Interface
This interface defines five methods. If the mouse is pressed and released at the same point, `mouseClicked()` is invoked. When the mouse enters a component, the...
mouseEntered( ) method is called. When it leaves, mouseExited( ) is called. The mousePressed( ) and mouseReleased( ) methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

```java
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

The MouseMotionListener Interface
This interface defines two methods. The mouseDragged( ) method is called multiple times as the mouse is dragged. The mouseMoved( ) method is called multiple times as the mouse is moved. Their general forms are shown here:

```java
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

The MouseWheelListener Interface
This interface defines the mouseWheelMoved( ) method that is invoked when the mouse wheel is moved. Its general form is shown here:

```java
void mouseWheelMoved(MouseWheelEvent mwe)
```
The TextListener Interface
This interface defines the textChanged( ) method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

    void textChanged(TextEvent te)

The WindowFocusListener Interface
This interface defines two methods: windowGainedFocus( ) and windowLostFocus( ). These are called when a window gains or loses input focus. Their general forms are shown here:

    void windowGainedFocus(WindowEvent we)
    void windowLostFocus(WindowEvent we)

The WindowListener Interface
This interface defines seven methods. The windowActivated( ) and windowDeactivated( ) methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified( ) method is called. When a window is deiconified, the windowDeiconified( ) method is called. When a window is opened or closed, the windowOpened( ) or windowClosed( ) methods are called, respectively. The windowClosing( ) method is called when a window is being closed. The general forms of these methods are
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)

Using the Delegation Event Model
Now that you have learned the theory behind the delegation event model and have had an overview of its various components, it is time to see it in practice. Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.

2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

To see how the delegation model works in practice, we will look at examples that handle two commonly
used event generators: the mouse and keyboard.

Handling Mouse Events

To handle mouse events, you must implement the MouseListener and the MouseMotionListener interfaces. (You may also want to implement MouseWheelListener, but we won’t be doing so, here.) The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet’s status window. Each time a button is pressed, the word “Down” is displayed at the location of the mouse pointer. Each time the button is released, the word “Up” is shown. If a button is clicked, the message “Mouse clicked” is displayed in the upper-left corner of the applet display area.

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, mouseX and mouseY, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by paint( ) to display output at the point of these occurrences.
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
   <applet code="MouseEvents" width=300 height=100>
   </applet>
*/

public class MouseEvents extends Applet
   implements MouseListener, MouseMotionListener {

   String msg = "";
   int mouseX = 0, mouseY = 0; // coordinates of mouse

   public void init() {
      addMouseListener(this);
      addMouseMotionListener(this);
   }

   // Handle mouse clicked.
   public void mouseClicked(MouseEvent me) {
      // save coordinates
      mouseX = 0;
      mouseY = 10;
      msg = "Mouse clicked."
      repaint();
   }
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse entered."
    repaint();
}

// Handle mouse exited.
public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse exited."
    repaint();
}

// Handle button pressed.
public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down"
    repaint();
}

// Handle button released.
public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up"
    repaint();
}
Let's look closely at this example. The MouseEvents class extends Applet and implements both the MouseListener and MouseMotionListener interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events.
events. This works because Component, which supplies the `addMouseListener()` and `addMouseMotionListener()` methods, is a superclass of Applet. Being both the source and the listener for events is a common situation for applets.

Inside `init()`, the applet registers itself as a listener for mouse events. This is done by using `addMouseListener()` and `addMouseMotionListener()`, which, as mentioned, are members of Component. They are shown here:

```java
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

Here, `ml` is a reference to the object receiving mouse events, and `mml` is a reference to the object receiving mouse motion events. In this program, the same object is used for both.

The applet then implements all of the methods defined by the `MouseListener` and `MouseMotionListener` interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Handling Keyboard Events

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that
Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a
KEY_PRESSED event is generated. This results in a call to the keyPressed() event handler. When the key is
released, a KEY_RELEASED event is generated and the keyReleased() handler is executed. If a character is
generated by the keystroke, then a KEY_TYPED event is sent and the keyTyped() handler is invoked. Thus, each
time the user presses a key, at least two and often three events are generated. If all you care about are actual
characters, then you can ignore the information passed by the key press and release events. However, if your
program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the
keyPressed() handler.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the
pressed/released status of each key in the status window.
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
  <applet code="SimpleKey" width=300 height=100>
</applet>
*/

public class SimpleKey extends Applet
    implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // output coordinates

    public void init() {
        addKeyListener(this);
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");
    }

    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up");
    }

    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }

    // Display keystrokes.
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}

Sample output is shown here:
If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the keyPressed() handler. They are not available through keyTyped().

To identify the keys, you use their virtual key codes. For example, the next applet outputs the name of a few of the special keys:

```java
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class KeyEvents extends Applet
    implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // output coordinates

    public void init() {
        addKeyListener(this);
    }
```
public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");

    int key = ke.getKeyCode();
    switch (key) {
    case KeyEvent.VK_F1:
        msg += "<F1>";
        break;
    case KeyEvent.VK_F2:
        msg += "<F2>";
        break;
    case KeyEvent.VK_F3:
        msg += "<F3>";
        break;
    case KeyEvent.VK_PAGE_DOWN:
        msg += "<PgDn>";
        break;
    case KeyEvent.VK_PAGE_UP:
        msg += "<PgUp>";
        break;
    case KeyEvent.VK_LEFT:
        msg += "<Left Arrow>";
        break;
    case KeyEvent.VK_RIGHT:
        msg += "<Right Arrow>";
        break;
    }

    repaint();
}

public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
}

public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
}

// Display keystrokes.
public void paint(Graphics g) {  
    g.drawString(msg, X, Y);
}
Sample output is shown here:

The procedures shown in the preceding keyboard and mouse event examples can be generalized to any type of event handling, including those events generated by controls. In later chapters, you will see many examples that handle other types of events, but they will all follow the same basic structure as the programs just described.

Adapter Classes
Java provides a special feature, called an adapter class, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the MouseMotionAdapter class has two
methods, `mouseDragged()` and `mouseMoved()`, which are the methods defined by the `MouseMotionListener` interface. If you were interested in only mouse drag events, then you could simply extend `MouseMotionAdapter` and override `mouseDragged()`. The empty implementation of `mouseMoved()` would handle the mouse motion events for you.

Table 23-4 lists the commonly used adapter classes in `java.awt.event` and notes the interface that each implements.

<table>
<thead>
<tr>
<th>Adapter Class</th>
<th>Listener 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>

Table 23-4 Commonly Used Listener Interfaces Implemented by Adapter Classes

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. `AdapterDemo` extends `Applet`. Its `init()` method creates an instance of `MyMouseAdapter` and registers that object to receive notifications of mouse events. It also creates an instance
of MyMouseMotionAdapter and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument.

MyMouseAdapter extends MouseAdapter and overrides the mouseClicked( ) method. The other mouse events are silently ignored by code inherited from the MouseAdapter class. MyMouseMotionAdapter extends MouseMotionAdapter and overrides the mouseDragged( ) method. The other mouse motion event is silently ignored by code inherited from the MouseMotionAdapter class.

Note that both of the event listener classes save a reference to the applet. This information is provided as an argument to their constructors and is used later to invoke the showStatus( ) method.
As you can see by looking at the program, not having to implement all of the methods defined by the MouseMotionListener and MouseListener interfaces saves you a considerable amount of effort and prevents your
code from becoming cluttered with empty methods. As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a KeyAdapter.

Inner Classes

In Chapter 7, the basics of inner classes were explained. Here you will see why they are important. Recall that an inner class is a class defined within another class, or even within an expression. This section illustrates how inner classes can be used to simplify the code when using event adapter classes.

To understand the benefit provided by inner classes, consider the applet shown in the following listing. It does not use an inner class. Its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. MousePressedDemo extends Applet, and MyMouseAdapter extends MouseAdapter. The init( ) method of MousePressedDemo instantiates MyMouseAdapter and provides this object as an argument to the addMouseListener( ) method.

Notice that a reference to the applet is supplied as an argument to the MyMouseAdapter constructor. This reference is stored in an instance variable for later use by the mousePressed( ) method. When the mouse is pressed, it invokes the showStatus( ) method of the applet.
through the stored applet reference. In other words, 
showStatus( ) is invoked relative to the applet reference 
stored by MyMouseAdapter.

```java
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;

/*
   <applet code="MousePressedDemo" width=200 height=100>
   </applet>
*/
public class MousePressedDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me) {
        mousePressedDemo.showStatus("Mouse Pressed.");
    }
}

The following listing shows how the preceding 
program can be improved by using an inner class. Here, 
InnerClassDemo is a top-level class that extends Applet. 
MyMouseAdapter is an inner class that extends MouseAdapter. Because MyMouseAdapter is defined 
within the scope of InnerClassDemo, it has access to all of the variables and methods within the scope of that 
class. Therefore, the mousePressed( ) method can call the 
showStatus( ) method directly. It no longer needs to do 
this via a stored reference to the applet. Thus, it is no
longer necessary to pass MyMouseAdapter() a reference to the invoking object.

Anonymous Inner Classes

An anonymous inner class is one that is not assigned a name. This section illustrates how an anonymous inner class can facilitate the writing of event handlers. Consider the applet shown in the following listing. As before, its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed.

```java
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;

/*
   <applet code="AnonymousInnerClassDemo" width=200 height=100>
    </applet>
*/

public class AnonymousInnerClassdemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter());
    }

    class MyMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent me) {
            showStatus("Mouse Pressed");
        }
    }
}
```
There is one top-level class in this program: AnonymousInnerClassDemo. The init( ) method calls the addMouseListener( ) method. Its argument is an expression that defines and instantiates an anonymous inner class. Let’s analyze this expression carefully.

The syntax new MouseAdapter(){…} indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends MouseAdapter. This new class is not named, but it is automatically instantiated when this expression is executed.

Because this anonymous inner class is defined within the scope of AnonymousInnerClassDemo, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the showStatus( ) method directly.

As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.
The Abstract Window Toolkit (AWT) was introduced in Chapter 22, where it was used in several example applets. This chapter begins its in-depth examination. The AWT contains numerous classes and methods that allow you to create and manage windows. It is also the foundation upon which Swing is built. The AWT is quite large and a full description would easily fill an entire book. Therefore, it is not possible to describe in detail every AWT class, method, or instance variable. However, this and the following two chapters explain the basic techniques needed to use the AWT effectively when creating your own applets or stand-alone GUI-based applications. From there, you will be able to explore other parts of the AWT on your own. You will also be ready to move on to Swing.

In this chapter, you will learn how to create and manage windows, manage fonts, output text, and utilize graphics. Chapter 25 describes the various controls, such as scroll bars and push buttons, supported by the AWT. It also explains further aspects of Java’s event handling mechanism. Chapter 26 examines the AWT’s imaging subsystem and animation.

Although a common use of the AWT is in applets, it is also used to create stand-alone windows that run in a
GUI environment, such as Windows. For the sake of convenience, most of the examples in this chapter are contained in applets. To run them, you need to use an applet viewer or a Java-compatible web browser. A few examples will demonstrate the creation of stand-alone, windowed programs.

One other point before beginning. Today, most Java programs employ user interfaces based on Swing. Because Swing provides richer implementations than does the AWT of some common GUI controls, such as buttons, lists, and check boxes, it is easy to jump to the conclusion that the AWT is no longer important, that it has been superseded by Swing. This assumption is, however, quite wrong. As mentioned, Swing is built on top of the AWT. Thus, many aspects of the AWT are also aspects of Swing. Furthermore, many AWT classes are used either directly or indirectly by Swing. Finally, for some types of small programs (especially small applets) that make only minimal use of a GUI, using the AWT rather than Swing still makes sense. Therefore, even though most interfaces today will be based on Swing, a solid knowledge of the AWT is still required. Simply put, you can’t be a great Java programmer without knowing the AWT.

NOTE

If you have not yet read Chapter 23, please do so now. It provides an overview of event handling,
which is used by many of the examples in this chapter.

AWT Classes

The AWT classes are contained in the java.awt package. It is one of Java’s largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table 24-1 lists some of the many AWT classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWTEvent</td>
<td>Encapsulates AWT events.</td>
</tr>
<tr>
<td>AWTEventMulticaster</td>
<td>Dispatches events to multiple listeners.</td>
</tr>
<tr>
<td>BorderLayout</td>
<td>The border layout manager. Border layouts use five components: North, South, East, West, and Center.</td>
</tr>
<tr>
<td>Button</td>
<td>Creates a push button control.</td>
</tr>
<tr>
<td>Canvas</td>
<td>A blank, semantics-free window.</td>
</tr>
<tr>
<td>CardLayout</td>
<td>The card layout manager. Card layouts emulate index cards. Only the one on top is showing.</td>
</tr>
<tr>
<td>Checkbox</td>
<td>Creates a check box control.</td>
</tr>
<tr>
<td>CheckboxGroup</td>
<td>Creates a group of check box controls.</td>
</tr>
<tr>
<td>CheckboxMenuItem</td>
<td>Creates an on/off menu item.</td>
</tr>
<tr>
<td>Choice</td>
<td>Creates a pop-up list.</td>
</tr>
<tr>
<td>Color</td>
<td>Manages colors in a portable, platform-independent fashion.</td>
</tr>
<tr>
<td>Component</td>
<td>An abstract superclass for various AWT components.</td>
</tr>
<tr>
<td>Container</td>
<td>A subclass of Component that can hold other components.</td>
</tr>
<tr>
<td>Cursor</td>
<td>Encapsulates a bitmapped cursor.</td>
</tr>
<tr>
<td>Dialog</td>
<td>Creates a dialog window.</td>
</tr>
<tr>
<td>Dimension</td>
<td>Specifies the dimensions of an object. The width is stored in width, and the height is stored in height.</td>
</tr>
<tr>
<td>EventQueue</td>
<td>Queues events.</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FileDialog</td>
<td>Creates a window from which a file can be selected.</td>
</tr>
<tr>
<td>FlowLayout</td>
<td>The flow layout manager. Flow layout positions components left to right, top to bottom.</td>
</tr>
<tr>
<td>Font</td>
<td>Encapsulates a type font.</td>
</tr>
<tr>
<td>FontMetrics</td>
<td>Encapsulates various information related to a font. This information helps you display text in a window.</td>
</tr>
<tr>
<td>Frame</td>
<td>Creates a standard window that has a title bar, resize corners, and a menu bar.</td>
</tr>
<tr>
<td>Graphics</td>
<td>Encapsulates the graphics context. This context is used by the various output methods to display output in a window.</td>
</tr>
<tr>
<td>GraphicsDevice</td>
<td>Describes a graphics device such as a screen or printer.</td>
</tr>
<tr>
<td>GraphicsEnvironment</td>
<td>Describes the collection of available Font and GraphicsDevice objects.</td>
</tr>
<tr>
<td>GridBagConstraints</td>
<td>Defines various constraints relating to the GridBagLayout class.</td>
</tr>
<tr>
<td>GridBagLayout</td>
<td>The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints.</td>
</tr>
<tr>
<td>GridLayout</td>
<td>The grid layout manager. Grid layout displays components in a two-dimensional grid.</td>
</tr>
<tr>
<td>Image</td>
<td>Encapsulates graphical images.</td>
</tr>
<tr>
<td>Insets</td>
<td>Encapsulates the borders of a container.</td>
</tr>
<tr>
<td>Label</td>
<td>Creates a label that displays a string.</td>
</tr>
<tr>
<td>List</td>
<td>Creates a list from which the user can choose. Similar to the standard Windows list box.</td>
</tr>
<tr>
<td>MediaTracker</td>
<td>Manages media objects.</td>
</tr>
</tbody>
</table>
Table 24-1 A Sampling of AWT Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu</td>
<td>Creates a pull-down menu.</td>
</tr>
<tr>
<td>MenuBar</td>
<td>Creates a menu bar.</td>
</tr>
<tr>
<td>MenuComponent</td>
<td>An abstract class implemented by various menu classes.</td>
</tr>
<tr>
<td>MenuItem</td>
<td>Creates a menu item.</td>
</tr>
<tr>
<td>MenuShortcut</td>
<td>Encapsulates a keyboard shortcut for a menu item.</td>
</tr>
<tr>
<td>Panel</td>
<td>The simplest concrete subclass of <strong>Container</strong>.</td>
</tr>
<tr>
<td>Point</td>
<td>Encapsulates a Cartesian coordinate pair, stored in $x$ and $y$.</td>
</tr>
<tr>
<td>Polygon</td>
<td>Encapsulates a polygon.</td>
</tr>
<tr>
<td>PopupMenu</td>
<td>Encapsulates a pop-up menu.</td>
</tr>
<tr>
<td>PrintJob</td>
<td>An abstract class that represents a print job.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Encapsulates a rectangle.</td>
</tr>
<tr>
<td>Robot</td>
<td>Supports automated testing of AWT-based applications.</td>
</tr>
<tr>
<td>Scrollbar</td>
<td>Creates a scroll bar control.</td>
</tr>
<tr>
<td>ScrollPane</td>
<td>A container that provides horizontal and/or vertical scroll bars for another component.</td>
</tr>
<tr>
<td>SystemColor</td>
<td>Contains the colors of GUI widgets such as windows, scroll bars, text, and others.</td>
</tr>
<tr>
<td>TextArea</td>
<td>Creates a multiline edit control.</td>
</tr>
<tr>
<td>TextComponent</td>
<td>A superclass for <strong>TextArea</strong> and <strong>TextField</strong>.</td>
</tr>
<tr>
<td>TextField</td>
<td>Creates a single-line edit control.</td>
</tr>
<tr>
<td>Toolkit</td>
<td>Abstract class implemented by the AWT.</td>
</tr>
<tr>
<td>Window</td>
<td>Creates a window with no frame, no menu bar, and no title.</td>
</tr>
</tbody>
</table>

Although the basic structure of the AWT has been the same since Java 1.0, some of the original methods were deprecated and replaced by new ones. For backward-compatibility, Java still supports all the original 1.0 methods. However, because these methods are not for use with new code, this book does not describe them.

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from
Panel, which is used by applets, and those derived from Frame, which creates a standard application window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 24-1 shows the class hierarchy for Panel and Frame. Let’s look at each of these classes now.

Component

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a visual component. Except for menus, all user interface elements that are displayed on the screen and that interact with the user are subclasses of Component. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. (You already used many of these methods when you created applets in Chapters 22 and 23.) A Component object is responsible for remembering the current foreground and background colors and the currently selected text font.
Container

The Container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a Container (since they are themselves instances of Component). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers, which you will learn about in Chapter 25.

Panel

The Panel class is a concrete subclass of Container. A Panel may be thought of as a recursively nestable, concrete screen component. Panel is the superclass for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object. In essence, a
Panel is a window that does not contain a title bar, menu bar, or border. This is why you don’t see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a Panel object by its add() method (inherited from Container). Once these components have been added, you can position and resize them manually using the setLocation(), setSize(), setPreferredSize(), or setBounds() methods defined by Component.

Window
The Window class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop. Generally, you won’t create Window objects directly. Instead, you will use a subclass of Window called Frame, described next.

Frame
Frame encapsulates what is commonly thought of as a “window.” It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.

Canvas
Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: Canvas. Canvas encapsulates a blank window upon which you can draw. You will see
Working with Frame Windows

After the applet, the type of window you will most often create is derived from Frame. You will use it to create child windows within applets, and top-level or child windows for stand-alone applications. As mentioned, it creates a standard-style window.

Here are two of Frame’s constructors:

Frame( ) throws HeadlessException
Frame(String title) throws HeadlessException

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created. A HeadlessException is thrown if an attempt is made to create a Frame instance in an environment that does not support user interaction.

There are several key methods you will use when working with Frame windows. They are examined here.

Setting the Window’s Dimensions

The setSize( ) method is used to set the dimensions of the window. Its signature is shown here:

void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
The new size of the window is specified by newWidth and newHeight, or by the width and height fields of the Dimension object passed in newSize. The dimensions are specified in terms of pixels.

The getSize( ) method is used to obtain the current size of a window. One of its forms is shown here:

```java
Dimension getSize( )
```

This method returns the current size of the window contained within the width and height fields of a Dimension object.

Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call setVisible( ). Its signature is shown here:

```java
void setVisible(boolean visibleFlag)
```

The component is visible if the argument to this method is true. Otherwise, it is hidden.

Setting a Window’s Title

You can change the title in a frame window using setTitle( ), which has this general form:

```java
void setTitle(String newTitle)
```

Here, newTitle is the new title for the window.
Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed, by calling `setVisible(false)`. To intercept a window-close event, you must implement the `windowClosing()` method of the `WindowListener` interface. Inside `windowClosing()`, you must remove the window from the screen. The example in the next section illustrates this technique.

Creating a Frame Window in an Applet

While it is possible to simply create a window by creating an instance of `Frame`, you will seldom do so, because you will not be able to do much with it. For example, you will not be able to receive or process events that occur within it or easily output information to it. Most of the time, you will create a subclass of `Frame`. Doing so lets you override `Frame`’s methods and provide event handling.

Creating a new frame window from within an applet is actually quite easy. First, create a subclass of `Frame`. Next, override any of the standard applet methods, such as `init()`, `start()`, and `stop()`, to show or hide the frame as needed. Finally, implement the `windowClosing()` method of the `WindowListener` interface, calling `setVisible(false)` when the window is closed.

Once you have defined a `Frame` subclass, you can create an object of that class. This causes a frame
window to come into existence, but it will not be initially visible. You make it visible by calling setVisible(). When created, the window is given a default height and width. You can set the size of the window explicitly by calling the setSize() method.

The following applet creates a subclass of Frame called SampleFrame. A window of this subclass is instantiated within the init() method of AppletFrame. Notice that SampleFrame calls Frame’s constructor. This causes a standard frame window to be created with the title passed in title. This example overrides the applet’s start() and stop() methods so that they show and hide the child window, respectively. This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
   <applet code="AppletFrame" width=300 height=50>
    <applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);

        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // register it to receive those events
        addWindowListener(adapter);
    }
    public void paint(Graphics g) {
        g.drawString("This is in frame window", 10, 40);
    }
}
class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;

    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }

    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Create frame window.
public class AppletFrame extends Applet {
    Frame f;

    public void init() {
        f = new SampleFrame("A Frame Window");
        f.setSize(250, 250);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }

    public void paint(Graphics g) {
        g.drawString("This is in applet window", 10, 20);
    }
}

Sample output from this program is shown here:
Handling Events in a Frame Window

Since Frame is a subclass of Component, it inherits all the capabilities defined by Component. This means that you can use and manage a frame window just like you manage an applet’s main window. For example, you can override paint( ) to display output, call repaint( ) when you need to restore the window, and add event handlers. Whenever an event occurs in a window, the event handlers defined by that window will be called. Each window handles its own events. For example, the following program creates a window that responds to mouse events. The main applet window also responds to mouse events. When you experiment with this program, you will see that mouse events are sent to the window in which the event occurs.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;


/*
   <applet code="WindowEvents" width=300 height=50>
   </applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame
    implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX=10, mouseY=40;
    int movX=0, movY=0;

    SampleFrame(String title) {
        super(title);
        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // register it to receive those events
        addWindowListener(adapter);
    }
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
}

// Handle mouse entered.
public void mouseEntered(MouseEvent evtObj) {
    // save coordinates
    mouseX = 10;
    mouseY = 54;
    msg = "Mouse just entered child.";
    repaint();
}

// Handle mouse exited.
public void mouseExited(MouseEvent evtObj) {
    // save coordinates
    mouseX = 10;
    mouseY = 54;
    msg = "Mouse just left child window.";
    repaint();
}

// Handle mouse pressed.
public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}
public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
}

// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "*";
    repaint();
}

// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
    // save coordinates
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 60);
}
public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;

    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }
    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Applet window.
public class WindowEvents extends Applet
    implements MouseListener, MouseMotionListener {

    SampleFrame f;
    String msg = "";
    int mouseX=0, mouseY=10;
    int movX=0, movY=0;
// Create a frame window.
public void init() {
    f = new SampleFrame("Handle Mouse Events");
    f.setSize(300, 200);
    f.setVisible(true);

    // register this object to receive its own mouse events
    addMouseListener(this);
    addMouseMotionListener(this);
}

// Remove frame window when stopping applet.
public void stop() {
    f.setVisible(false);
}

// Show frame window when starting applet.
public void start() {
    f.setVisible(true);
}

// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just entered applet window.";
    repaint();
}

// Handle mouse exited.
public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
    repaint();
}

// Handle button pressed.
public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}
Sample output from this program is shown here:
Creating a Windowed Program

Although creating applets is a common use for Java’s AWT, it is also possible to create stand-alone AWT-based applications. To do this, simply create an instance of the window or windows you need inside `main()`. For example, the following program creates a frame window that responds to mouse clicks and keystrokes:
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Create a frame window.
public class AppWindow extends Frame {
    String keymsg = "This is a test.";
    String mousemsg = "";
    int mouseX = 30, mouseY = 30;

    public AppWindow() {
        addKeyListener(new MyKeyAdapter(this));
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }

    public void paint(Graphics g) {
        g.drawString(keymsg, 10, 40);
        g.drawString(mousemsg, mouseX, mouseY);
    }

    // Create the window.
    public static void main(String args[]) {
        AppWindow appwin = new AppWindow();

        appwin.setSize(new Dimension(300, 200));
        appwin.setTitle("An AWT-Based Application");
        appwin.setVisible(true);
    }
}
class MyKeyAdapter extends KeyAdapter {
    AppWindow appWindow;

    public MyKeyAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void keyTyped(KeyEvent ke) {
        appWindow.keymsg += ke.getKeyChar();
        appWindow.repaint();
    }
}

class MyMouseAdapter extends MouseAdapter {
    AppWindow appWindow;

    public MyMouseAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void mousePressed(MouseEvent me) {
        appWindow.mouseX = me.getX();
        appWindow.mouseY = me.getY();
        appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX + ", " + appWindow.mouseY;
        appWindow.repaint();
    }
}

class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}
Once created, a frame window takes on a life of its own. Notice that `main()` ends with the call to `appwin.setVisible(true)`. However, the program keeps running until you close the window. In essence, when creating a windowed application, you will use `main()` to launch its top-level window. After that, your program will function as a GUI-based application, not like the console-based programs used earlier.

Displaying Information Within a Window

In the most general sense, a window is a container for information. Although we have already output small amounts of text to a window in the preceding examples, we have not begun to take advantage of a window’s ability to present high-quality text and graphics. Indeed, much of the power of the AWT comes from its support for these items. For this reason, the remainder of this chapter discusses Java’s text-, graphics-, and font-handling capabilities. As you will see, they are both powerful and flexible.
Working with Graphics

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a graphics context. A graphics context is encapsulated by the Graphics class and is obtained in two ways:

• It is passed to a method, such as paint( ) or update( ), as an argument.
• It is returned by the getGraphics( ) method of Component.

For the sake of convenience the remainder of the examples in this chapter will demonstrate graphics in a main applet window. However, the same techniques will apply to any other window.

The Graphics class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Let’s take a look at several of the drawing methods.

Drawing Lines

Lines are drawn by means of the drawLine( ) method,
shown here:

g void drawLine(int startX, int startY, int endX, int endY)

drawLine( ) displays a line in the current drawing color that begins at startX, startY and ends at endX, endY.

The following applet draws several lines:

```java
// Draw lines
import java.awt.*;
import java.applet.*;
/
<applet code="Lines" width=300 height=200>
</applet>
*/

public class Lines extends Applet {
    public void paint(Graphics g) {
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);
    }
}
```

Sample output from this program is shown here:
Drawing Rectangles

The `drawRect()` and `fillRect()` methods display an outlined and filled rectangle, respectively. They are shown here:

```java
void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)
```

The upper-left corner of the rectangle is at `top`, `left`. The dimensions of the rectangle are specified by `width` and `height`.

To draw a rounded rectangle, use `drawRoundRect()` or `fillRoundRect()`, both shown here:

```java
void drawRoundRect(int top, int left, int width, int height,
                    int xDiam, int yDiam)
```
void fillRoundRect(int top, int left, int width, int height,
                   int xDiam, int yDiam)

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at top, left. The dimensions of the rectangle are specified by width and height. The diameter of the rounding arc along the X axis is specified by xDiam. The diameter of the rounding arc along the Y axis is specified by yDiam.

The following applet draws several rectangles:

```java
// Draw rectangles
import java.awt.*;
import java.applet.*;

/*
 <applet code="Rectangles" width=300 height=200>
 </applet>
 */

class Rectangles extends Applet {
   public void paint(Graphics g) {
      g.drawRect(10, 10, 60, 50);
      g.fillRect(100, 10, 60, 50);
      g.drawRoundRect(190, 10, 60, 50, 15, 15);
      g.fillRoundRect(70, 90, 140, 100, 30, 40);
   }
}
```

Sample output from this program is shown here:
Drawing Ellipses and Circles

To draw an ellipse, use `drawOval()` . To fill an ellipse, use `fillOval()` . These methods are shown here:

```java
void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)
```

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by `top`, `left` and whose width and height are specified by `width` and `height` . To draw a circle, specify a square as the bounding rectangle.

The following program draws several ellipses:

```java
// Draw Ellipses
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/
```
public class Ellipses extends Applet {
    public void paint(Graphics g) {
        g.drawOval(10, 10, 50, 50);
        g.fillOval(100, 10, 75, 50);
        g.drawOval(190, 10, 90, 30);
        g.fillOval(70, 90, 140, 100);
    }
}

Sample output from this program is shown here:

![Applet Viewer: Ellipses](image)

Drawing Arcs

Arcs can be drawn with `drawArc()` and `fillArc()`, shown here:

```java
void drawArc(int top, int left, int width, int height, int startAngle,
             int sweepAngle)

void fillArc(int top, int left, int width, int height, int startAngle,
             int sweepAngle)
```
The arc is bounded by the rectangle whose upper-left corner is specified by top, left and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degrees. Zero degrees is on the horizontal, at the three o’clock position. The arc is drawn counterclockwise if sweepAngle is positive, and clockwise if sweepAngle is negative. Therefore, to draw an arc from twelve o’clock to six o’clock, the start angle would be 90 and the sweep angle 180.

The following applet draws several arcs:

```java
// Draw Arcs
import java.awt.*;
import java.applet.*;
/
<applet code="Arcs" width=300 height=200>
</applet>
*/

public class Arcs extends Applet {
    public void paint(Graphics g) {
        g.drawArc(10, 40, 70, 70, 0, 75);
        g.fillArc(100, 40, 70, 70, 0, 75);
        g.drawArc(10, 100, 70, 80, 0, 175);
        g.fillArc(100, 100, 70, 90, 0, 270);
        g.drawArc(200, 80, 80, 80, 0, 180);
    }
}
```

Sample output from this program is shown here:
Drawing Polygons

It is possible to draw arbitrarily shaped figures using `drawPolygon()` and `fillPolygon()`, shown here:

```java
void drawPolygon(int x[ ], int y[ ], int numPoints)
void fillPolygon(int x[ ], int y[ ], int numPoints)
```

The polygon’s endpoints are specified by the coordinate pairs contained within the `x` and `y` arrays. The number of points defined by `x` and `y` is specified by `numPoints`. There are alternative forms of these methods in which the polygon is specified by a `Polygon` object.

The following applet draws an hourglass shape:

```java
// Draw Polygon
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/
```
public class HourGlass extends Applet {
    public void paint(Graphics g) {
        int xpoints[] = {30, 200, 30, 200, 30};
        int ypoints[] = {30, 30, 200, 200, 30};
        int num = 5;

        g.drawPolygon(xpoints, ypoints, num);
    }
}

Sample output from this program is shown here:

Sizing Graphics
Often, you will want to size a graphics object to fit the current size of the window in which it is drawn. To do so, first obtain the current dimensions of the window by calling getSize( ) on the window object. It returns the dimensions of the window encapsulated within a Dimension object. Once you have the current size of the window, you can scale your graphical output accordingly.
To demonstrate this technique, here is an applet that will start as a 200x200-pixel square and grow by 25 pixels in width and height with each mouse click until the applet gets larger than 500x500. At that point, the next click will return it to 200x200, and the process starts over.

Within the window, a rectangle is drawn around the inner border of the window; within that rectangle, an X is drawn so that it fills the window. This applet works in appletviewer, but it may not work in a browser window.
Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is
encapsulated by the Color class.

As you saw in Chapter 22, Color defines several constants (for example, Color.black) to specify a number of common colors. You can also create your own colors, using one of the color constructors. Three commonly used forms are shown here:

- `Color(int red, int green, int blue)
- `Color(int rgbValue)
- `Color(float red, float green, float blue)

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```java
new Color(255, 100, 100); // light red
```

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```java
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

The final constructor, `Color(float, float, float)`, takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the `setForeground()` and `setBackground()` methods.
described in Chapter 22. You can also select it as the current drawing color.

Color Methods

The Color class defines several methods that help manipulate colors. Several are examined here.

Using Hue, Saturation, and Brightness

The hue-saturation-brightness (HSB) color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, hue is a wheel of color. The hue can be specified with a number between 0.0 and 1.0, which is used to obtain an angle into the color wheel. (The principal colors are approximately red, orange, yellow, green, blue, indigo, and violet.) Saturation is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. Brightness values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. Color supplies two methods that let you convert between RGB and HSB. They are shown here:

```java
static int HSBtoRGB(float hue, float saturation, float brightness)
static float[] RGBtoHSB(int red, int green, int blue, float values[])
```

HSBtoRGB( ) returns a packed RGB value compatible with the Color(int) constructor. RGBtoHSB( ) returns a float array of HSB values corresponding to RGB integers. If values is not null, then this array is given the HSB
values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed( ), getGreen( ), getBlue( )
You can obtain the red, green, and blue components of a color independently using getRed( ), getGreen( ), and getBlue( ), shown here:

```java
int getRed( )
int getGreen( )
int getBlue( )
```

Each of these methods returns the RGB color component found in the invoking Color object in the lower 8 bits of an integer.

getRGB( )
To obtain a packed, RGB representation of a color, use getRGB( ), shown here:

```java
int getRGB( )
```

The return value is organized as described earlier.

Setting the Current Graphics Color
By default, graphics objects are drawn in the current foreground color. You can change this color by calling the Graphics method setColor( ):
void setColor(Color newColor)

Here, newColor specifies the new drawing color.

You can obtain the current color by calling getColor( ), shown here:

    Color getColor( )

A Color Demonstration Applet

The following applet constructs several colors and draws various objects using these colors:
Setting the Paint Mode
The paint mode determines how objects are drawn in a window. By default, new output to a window overwrites any preexisting contents. However, it is possible to have
new objects XORed onto the window by using setXORMode(), as follows:

    void setXORMode(Color xorColor)

Here, xorColor specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over.

To return to overwrite mode, call setPaintMode(), shown here:

    void setPaintMode()

In general, you will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.
// Demonstrate XOR mode.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
   <applet code="XOR" width=400 height=200>
    </applet>
*/

public class XOR extends Applet {
    int chsX=100, chsY=100;

    public XOR() { 
        addMouseMotionListener(new MouseMotionAdapter() { 
            public void mouseMoved(MouseEvent me) { 
                int x = me.getX();
                int y = me.getY();
                chsX = x-10;
                chsY = y-10;
                repaint();
            }
        });
    }
}
public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.setColor(Color.blue);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.setColor(Color.green);
    g.drawRect(10, 10, 60, 50);
    g.fillRect(100, 10, 60, 50);
    g.setColor(Color.red);
    g.drawRectRoundRect(190, 10, 60, 50, 15, 15);
    g.fillRectRoundRect(70, 90, 140, 100, 30, 40);
    g.setColor(Color.cyan);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    // xor cross hairs
    g.setColor(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setColor(Color.black);
}

Sample output from this program is shown here:
The AWT supports multiple type fonts. Years ago, fonts emerged from the domain of traditional typesetting to become an important part of computer-generated documents and displays. The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts.

Fonts have a family name, a logical font name, and a face name. The family name is the general name of the font, such as Courier. The logical name specifies a category of font, such as Monospaced. The face name specifies a specific font, such as Courier Italic.

Fonts are encapsulated by the Font class. Several of the methods defined by Font are listed in Table 24-2.
Table 24-2 A Sampling of Methods Defined by Font

The Font class defines these variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>String name</td>
<td>Name of the font</td>
</tr>
<tr>
<td>float pointSize</td>
<td>Size of the font in points</td>
</tr>
<tr>
<td>int size</td>
<td>Size of the font in points</td>
</tr>
<tr>
<td>int style</td>
<td>Font style</td>
</tr>
</tbody>
</table>

Determining the Available Fonts

When working with fonts, often you need to know which
fonts are available on your machine. To obtain this information, you can use the `getAvailableFontFamilyNames()` method defined by the `GraphicsEnvironment` class. It is shown here:

```java
String[] getAvailableFontFamilyNames()
```

This method returns an array of strings that contains the names of the available font families.

In addition, the `getAllFonts()` method is defined by the `GraphicsEnvironment` class. It is shown here:

```java
Font[] getAllFonts()
```

This method returns an array of `Font` objects for all of the available fonts.

Since these methods are members of `GraphicsEnvironment`, you need a `GraphicsEnvironment` reference to call them. You can obtain this reference by using the `getLocalGraphicsEnvironment()` static method, which is defined by `GraphicsEnvironment`. It is shown here:

```java
static GraphicsEnvironment getLocalGraphicsEnvironment()
```

Here is an applet that shows how to obtain the names of the available font families:

```java
// Display Fonts
/*
<applet code="ShowFonts" width=550 height=60>
```
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
    public void paint(Graphics g) {
        String msg = "";
        String FontList[];

        GraphicsEnvironment ge =
          GraphicsEnvironment.getLocalGraphicsEnvironment();
        FontList = ge.getAvailableFontFamilyNames();
        for(int i = 0; i < FontList.length; i++)
            msg += FontList[i] + " ";

        g.drawString(msg, 4, 16);
    }
}

Sample output from this program is shown next. However, when you run this program, you may see a different list of fonts than the one shown in this illustration.

Creating and Selecting a Font
To select a new font, you must first construct a Font object that describes that font. One Font constructor has this general form:

    Font(String fontName, int fontStyle, int pointSize)
Here, fontName specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support the following fonts: Dialog, DialogInput, Sans Serif, Serif, and Monospaced. Dialog is the font used by your system’s dialog boxes. Dialog is also the default if you don’t explicitly set a font. You can also use any other fonts supported by your particular environment, but be careful—these other fonts may not be universally available.

The style of the font is specified by fontStyle. It may consist of one or more of these three constants: Font.PLAIN, Font.BOLD, and Font.ITALIC. To combine styles, OR them together. For example, Font.BOLD | Font.ITALIC specifies a bold, italics style.

The size, in points, of the font is specified by pointSize.

To use a font that you have created, you must select it using setFont(), which is defined by Component. It has this general form:

```java
void setFont(Font fontObj)
```
Here, fontObj is the object that contains the desired font.

The following program outputs a sample of each standard font. Each time you click the mouse within its window, a new font is selected and its name is displayed.

```java
// Show fonts.
import java.applet.*;
```
public class SampleFonts extends Applet {
    int next = 0;
    Font f;
    String msg;

    public void init() {
        f = new Font("Dialog", Font.PLAIN, 12);
        msg = "Dialog";
       setFont(f);
        addMouseListener(new MyMouseAdapter(this));
    }

    public void paint(Graphics g) {
        g.drawString(msg, 4, 20);
    }
}

class MyMouseAdapter extends MouseAdapter {
    SampleFonts sampleFonts;

    public MyMouseAdapter(SampleFonts sampleFonts) {
        this.sampleFonts = sampleFonts;
    }

    public void mousePressed(MouseEvent me) {
        // Switch fonts with each mouse click.
        sampleFonts.next++;
        switch(sampleFonts.next) {
        case 0:
            sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
            sampleFonts.msg = "Dialog";
            break;
        case 1:
            sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
            sampleFonts.msg = "DialogInput";
            break;
        case 2:
            sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
            sampleFonts.msg = "SansSerif";
            break;
        case 3:
            sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
            sampleFonts.msg = "Serif";
            break;
        case 4:
            sampleFonts.f = new Font("SansSerif", Font.BOLD, 12);
            sampleFonts.msg = "SansSerif Bold";
            break;
        case 5:
            sampleFonts.f = new Font("Dialog", Font.ITALIC, 12);
            sampleFonts.msg = "Dialog Italic";
            break;
        case 6:
            sampleFonts.f = new Font("DialogInput", Font.BOLD, 12);
            sampleFonts.msg = "DialogInput Bold";
            break;
        case 7:
            sampleFonts.f = new Font("SansSerif", Font.BOLD | Font.ITALIC, 12);
            sampleFonts.msg = "SansSerif Bold Italic";
            break;
        case 8:
            sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
            sampleFonts.msg = "Dialog";
            break;
        // Add more cases if necessary.
    }
}
case 3:
    sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
sampleFonts.msg = "Serif";
    break;

case 4:
    sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
sampleFonts.msg = "Monospaced";
    break;
}

if(sampleFonts.next == 4) sampleFonts.next = -1;

sampleFonts.setFont(sampleFonts.f);
sampleFonts.repaint();
}
}

Sample output from this program is shown here:

![Applet Viewer: SampleFonts](image)

Obtaining Font Information

Suppose you want to obtain information about the currently selected font. To do this, you must first get the current font by calling `getFont()` . This method is defined by the Graphics class, as shown here:

```java
Font getFont()
```

Once you have obtained the currently selected font, you can retrieve information about it using various methods defined by Font. For example, this applet displays the
name, family, size, and style of the currently selected font:

```java
// Display font info.
import java.applet.*;
import java.awt.*;
/
<applet code="FontInfo" width=350 height=60>
</applet>
*/

class FontInfo extends Applet {
    public void paint(Graphics g) {
        Font f = g.getFont();
        String fontName = f.getName();
        String fontFamily = f.getFamily();
        int fontSize = f.getSize();
        int fontStyle = f.getStyle();
        String msg = "Family: " + fontName;
        msg += ", Font: " + fontFamily;
        msg += ", Size: " + fontSize + "", Style: ";
        if((fontStyle & Font.BOLD) == Font.BOLD)
            msg += "Bold ";
        if((fontStyle & Font.ITALIC) == Font.ITALIC)
            msg += "Italic ";
        if((fontStyle & Font.PLAIN) == Font.PLAIN)
            msg += "Plain ";
        g.drawString(msg, 4, 16);
    }
}
```

Managing Text Output Using FontMetrics

As just explained, Java supports a number of fonts. For most fonts, characters are not all the same dimension—most fonts are proportional. Also, the height of each character, the length of descenders (the hanging parts of letters, such as y), and the amount of space between horizontal lines vary from font to font. Further, the point
size of a font can be changed. That these (and other) attributes are variable would not be of too much consequence except that Java demands that you, the programmer, manually manage virtually all text output.

Given that the size of each font may differ and that fonts may be changed while your program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font. For example, to write one line of text after another implies that you have some way of knowing how tall the font is and how many pixels are needed between lines. To fill this need, the AWT includes the FontMetrics class, which encapsulates various information about a font. Let’s begin by defining the common terminology used when describing fonts:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>The top-to-bottom size of a line of text</td>
</tr>
<tr>
<td>Baseline</td>
<td>The line that the bottoms of characters are aligned to (not counting descent)</td>
</tr>
<tr>
<td>Ascent</td>
<td>The distance from the baseline to the top of a character</td>
</tr>
<tr>
<td>Descent</td>
<td>The distance from the baseline to the bottom of a character</td>
</tr>
<tr>
<td>Leading</td>
<td>The distance between the bottom of one line of text and the top of the next</td>
</tr>
</tbody>
</table>

As you know, we have used the drawString( ) method in many of the previous examples. It paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods. It is a common error to draw a string at the same coordinate that you would draw a box. For example, if you were to draw a
rectangle at coordinate 0,0, you would see a full rectangle. If you were to draw the string “Typesetting” at 0,0, you would only see the tails (or descenders) of the y, p, and g. As you will see, by using font metrics, you can determine the proper placement of each string that you display.

FontMetrics defines several methods that help you manage text output. Several commonly used ones are listed in Table 24-3. These methods help you properly display text in a window. Let’s look at some examples.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int bytesWidth(byte b[], int start, int numBytes)</td>
<td>Returns the width of numBytes characters held in array b, beginning at start.</td>
</tr>
<tr>
<td>int charWidth(char c[], int start, int numChars)</td>
<td>Returns the width of numChars characters held in array c, beginning at start.</td>
</tr>
<tr>
<td>int charWidth(char c)</td>
<td>Returns the width of c.</td>
</tr>
<tr>
<td>int charWidth(int c)</td>
<td>Returns the width of c.</td>
</tr>
<tr>
<td>int getAscent()</td>
<td>Returns the ascent of the font.</td>
</tr>
<tr>
<td>int getDescent()</td>
<td>Returns the descent of the font.</td>
</tr>
<tr>
<td>Font getFont()</td>
<td>Returns the font.</td>
</tr>
<tr>
<td>int getHeight()</td>
<td>Returns the height of a line of text. This value can be used to output multiple lines of text in a window.</td>
</tr>
<tr>
<td>int getLeading()</td>
<td>Returns the space between lines of text.</td>
</tr>
<tr>
<td>int getMaxAdvance()</td>
<td>Returns the width of the widest character. –1 is returned if this value is not available.</td>
</tr>
<tr>
<td>int getMaxAscent()</td>
<td>Returns the maximum ascent.</td>
</tr>
<tr>
<td>int getMaxDescent()</td>
<td>Returns the maximum descent.</td>
</tr>
<tr>
<td>int[] getWeights()</td>
<td>Returns the widths of the first 256 characters.</td>
</tr>
<tr>
<td>int stringWidth(String str)</td>
<td>Returns the width of the string specified by str.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns the string equivalent of the invoking object.</td>
</tr>
</tbody>
</table>

Table 24-3 A Sampling of Methods Defined by FontMetrics
Displaying Multiple Lines of Text

Perhaps the most common use of FontMetrics is to determine the spacing between lines of text. The second most common use is to determine the length of a string that is being displayed. Here, you will see how to accomplish these tasks.

In general, to display multiple lines of text, your program must manually keep track of the current output position. Each time a newline is desired, the Y coordinate must be advanced to the beginning of the next line. Each time a string is displayed, the X coordinate must be set to the point at which the string ends. This allows the next string to be written so that it begins at the end of the preceding one.

To determine the spacing between lines, you can use the value returned by getLeading(). To determine the total height of the font, add the value returned by getAscent() to the value returned by getDescent(). You can then use these values to position each line of text you output. However, in many cases, you will not need to use these individual values. Often, all that you will need to know is the total height of a line, which is the sum of the leading space and the font’s ascent and descent values. The easiest way to obtain this value is to call getHeight(). Simply increment the Y coordinate by this value each time you want to advance to the next line when outputting text.

To start output at the end of previous output on the
same line, you must know the length, in pixels, of each string that you display. To obtain this value, call stringWidth( ). You can use this value to advance the X coordinate each time you display a line.

The following applet shows how to output multiple lines of text in a window. It also displays multiple sentences on the same line. Notice the variables curX and curY. They keep track of the current text output position.

```java
// Demonstrate multiline output.
import java.applet.*;
import java.awt.*;

/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/

public class MultiLine extends Applet {
    int curX=0, curY=0; // current position

    public void init() {
        Font f = new Font("SansSerif", Font.PLAIN, 12);
        setFont(f);
    }
```
public void paint(Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    nextLine("This is on line one.", g);
    nextLine("This is on line two.", g);
    sameLine(" This is on same line.", g);
    sameLine(" This, too.", g);
    nextLine("This is on line three.", g);
    curX = curY = 0; // Reset coordinates for each repaint.
}

// Advance to next line.
void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    curY += fm.getHeight();  // advance to next line
    curX = 0;
    g.drawString(s, curX, curY);
    curX = fm.stringWidth(s);  // advance to end of line
}

// Display on same line.
void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s);  // advance to end of line
}

Sample output from this program is shown here:

```
Applet
This is on line one.
This is on line two. This is on same line. This, too.
This is on line three.
```
Applet started.
Centering Text

Here is an example that centers text, left to right, top to bottom, in a window. It obtains the ascent, descent, and width of the string and computes the position at which it must be displayed to be centered.

```java
// Center text.
import java.applet.*;
import java.awt.*;
/*
   <applet code="CenterText" width=200 height=100>
     </applet>
*/

public class CenterText extends Applet {
    final Font f = new Font("SansSerif", Font.BOLD, 18);

    public void paint(Graphics g) {
        Dimension d = this.getSize();

        g.setColor(Color.white);
        g.fillRect(0, 0, d.width, d.height);
        g.setColor(Color.black);
        g.setFont(f);
        drawCenteredString("This is centered.", d.width, d.height,
                          0, 0, d.width-1, d.height-1);
    }

    public void drawCenteredString(String s, int w, int h,
                                   Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        int x = (w - fm.stringWidth(s)) / 2;
        int y = (fm.getAscent() + (h - (fm.getAscent()
                                      + fm.getDescent()))/2);
        g.drawString(s, x, y);
    }
}
```

Following is a sample output from this program:
Multiline Text Alignment

When using a word processor, it is common for text to be aligned so that one or more of the edges of the text make a straight line. For example, most word processors can left-justify and/or right-justify text. Most can also center text. In the following program, you will see how to accomplish these actions.

In the program, the string to be justified is broken into individual words. For each word, the program keeps track of its length in the current font and automatically advances to the next line if the word will not fit on the current line. Each completed line is displayed in the window in the currently selected alignment style. Each time you click the mouse in the applet’s window, the alignment style is changed. Sample output from this program is shown here:
Output to a Java window is actually quite easy. As you have seen, the AWT provides support for fonts, colors, text, and graphics.

Of course, you must effectively utilize these items if you are to achieve professional results.

Applet started.
Demonstrate text alignment.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/* <title>Text Layout</title>
 <applet code="TextLayout" width=200 height=200>
  <param name="text" value="Output to a Java window is actually quite easy."
  As you have seen, the AWT provides support for fonts, colors, text, and graphics. <P> Of course, you must effectively utilize these items if you are to achieve professional results.
  </param name="fontname" value="Serif">
  <param name="fontSize" value="14">
   </applet>
*/

public class TextLayout extends Applet {
   final int LEFT = 0;
   final int RIGHT = 1;
   final int CENTER = 2;
   final int LEFTRIGHT =3;
   int align;
   Dimension d;
   Font f;
   FontMetrics fm;
   int fontSize;
   int fh, bl;
   int space;
   String text;
public void init() {
    setBackground(Color.white);
    text = getParameter("text");
    try {
        fontSize = Integer.parseInt(getParameter("fontSize"));
    } catch (NumberFormatException e) {
        fontSize=14;
    }
    align = LEFT;
    addMouseListener(new MyMouseListener(this));
}

public void paint(Graphics g) {
    update(g);
}

public void update(Graphics g) {
    d = getSize();
    g.setColor(getBackground());
    g.fillRect(0, 0, d.width, d.height);
    if(f==null) f = new Font(getParameter("fontname"),
                              Font.PLAIN, fontSize);

    g.setFont(f);
    if(fm == null) {
        fm = g.getFontMetrics();
        bl = fm.getAscent();
        fh = bl + fm.getDescent();
        space = fm.stringWidth(" ");
    }
}
g.setColor(Color.black);
StringTokenizer st = new StringTokenizer(text);
int x = 0;
int nextx;
int y = 0;
String word, sp;
int wordCount = 0;
String line = "";
while (st.hasMoreTokens()) {
    word = st.nextToken();
    if (word.equals("<P>")) {
        drawString(g, line, wordCount,
                   fm.stringWidth(line), y+bl);
        line = "";
        wordCount = 0;
        x = 0;
        y = y + (fh * 2);
    }
    else {
        int w = fm.stringWidth(word);
        if(( nextx = (x+space+w)) > d.width ) {
            drawString(g, line, wordCount,
                       fm.stringWidth(line), y+bl);
            line = "";
            wordCount = 0;
            x = 0;
            y = y + fh;
        }
        if(x!=0) {sp = " ";} else {sp = "";}
        line = line + sp + word;
        x = x + space + w;
        wordCount++;
    }
}
drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
public void drawString(Graphics g, String line,
                        int wc, int lineW, int y) {
    switch (align) {
        case LEFT: g.drawString(line, 0, y);
                    break;
        case RIGHT: g.drawString(line, d.width-lineW, y);
                    break;
        case CENTER: g.drawString(line, (d.width-lineW)/2, y);
                    break;
        case LEFTRIGHT:
            if (lineW < (int)(d.width*.75)) {
                g.drawString(line, 0, y);
            }
    }
}
Let's take a closer look at how this applet works. The applet first creates several constants that will be used to determine the alignment style, and then declares several variables. The init( ) method obtains the text that will be displayed. It then initializes the font size in a try-catch block, which will set the font size to 14 if the fontSize parameter is missing from the HTML. The text parameter
is a long string of text, with the HTML tag <P> as a paragraph separator.

The update( ) method is the engine for this example. It sets the font and gets the baseline and font height from a font metrics object. Next, it creates a StringTokenizer and uses it to retrieve the next token (a string separated by whitespace) from the string specified by text. If the next token is <P>, it advances the vertical spacing. Otherwise, update( ) checks to see if the length of this token in the current font will go beyond the width of the column. If the line is full of text or if there are no more tokens, the line is output by a custom version of drawString( ).

The first three cases in drawString( ) are simple. Each aligns the string that is passed in line to the left or right edge or to the center of the column, depending upon the alignment style. The LEFTRIGHT case aligns both the left and right sides of the string. This means that we need to calculate the remaining whitespace (the difference between the width of the string and the width of the column) and distribute that space between each of the words. The last method in this class advances the alignment style each time you click the mouse on the applet’s window.
CHAPTER 25
Using AWT Controls, Layout Managers, and Menus

This chapter continues our exploration of the Abstract Window Toolkit (AWT). It begins with an examination of the standard controls and layout managers. It then discusses menus and the menu bar. The chapter also includes a discussion of two high-level components: the dialog box and the file dialog box. It concludes with another look at event handling.

Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

In addition to the controls, a frame window can also include a standard-style menu bar. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. A menu bar is always positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls.

While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task. For the first part of this
chapter, which introduces the various controls, the default layout manager will be used. This displays components in a container using left-to-right, top-to-bottom organization. Once the controls have been covered, the layout managers will be examined. There you will see how to better manage the positioning of your controls.

Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Editing

These controls are subclasses of Component.

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling add(), which is defined by Container. The add() method has several forms. The following form is the one that is used for the first part of this chapter:
Component add(Component compObj)

Here, compObj is an instance of the control that you want to add. A reference to compObj is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call remove(). This method is also defined by Container. Here is one of its forms:

void remove(Component obj)

Here, obj is a reference to the control you want to remove. You can remove all controls by calling removeAll().

Responding to Controls
Except for labels, which are passive, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 23, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

The HeadlessException
Most of the AWT controls described in this chapter have constructors that can throw a HeadlessException when an attempt is made to instantiate a GUI component in a noninteractive environment (such as one in which no display, mouse, or keyboard is present). The HeadlessException was added by Java 1.4. You can use this exception to write code that can adapt to noninteractive environments. (Of course, this is not always possible.) This exception is not handled by the programs in this chapter because an interactive environment is required to demonstrate the AWT controls.

Labels
The easiest control to use is a label. A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. Label defines the following constructors:

    Label() throws HeadlessException
    Label(String str) throws HeadlessException
    Label(String str, int how) throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by str. This string is left-justified. The third version creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER.
You can set or change the text in a label by using the `setText( )` method. You can obtain the current label by calling `getText( )`. These methods are shown here:

```java
void setText(String str)
String getText( )
```

For `setText( )`, `str` specifies the new label. For `getText( )`, the current label is returned.

You can set the alignment of the string within the label by calling `setAlignment( )`. To obtain the current alignment, call `getAlignment( )`. The methods are as follows:

```java
void setAlignment(int how)
int getAlignment( )
```

Here, `how` must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

```java
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/

public class LabelDemo extends Applet {
    public void init() {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");
```
// add labels to applet window
add(one);
add(two);
add(three);
}
}

Here is the window created by the LabelDemo applet. Notice that the labels are organized in the window by the default layout manager. Later, you will see how to control more precisely the placement of the labels.

Using Buttons
Perhaps the most widely used control is the push button. A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors:

    Button( ) throws HeadlessException
    Button(String str) throws HeadlessException

The first version creates an empty button. The second creates a button that contains str as a label.

After a button has been created, you can set its label
by calling `setLabel()`. You can retrieve its label by calling `getLabel()`. These methods are as follows:

```java
void setLabel(String str)
String getLabel()
```

Here, `str` becomes the new label for the button.

**Handling Buttons**

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the `ActionListener` interface. That interface defines the `actionPerformed()` method, which is called when an event occurs. An `ActionEvent` object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the action command string associated with the button. By default, the action command string is the label of the button. Usually, either the button reference or the action command string can be used to identify the button. (You will soon see examples of each approach.)

Here is an example that creates three buttons labeled “Yes”, “No”, and “Undecided”. Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is
obtained by calling the `getActionCommand()` method on the `ActionEvent` object passed to `actionPerformed()`. 

```java
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
   <applet code="ButtonDemo" width=250 height=150>
    </applet>
*/

public class ButtonDemo extends Applet implements ActionListener {
    String msg = "";
    Button yes, no, maybe;

    public void init() {
        yes = new Button("Yes");
        no = new Button("No");
        maybe = new Button("Undecided");

        add(yes);
        add(no);
        add(maybe);

        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        String str = ae.getActionCommand();
        if(str.equals("Yes")) {
            msg = "You pressed Yes. ";
        }
        else if(str.equals("No")) {
            msg = "You pressed No. ";
        }
        else {
            msg = "You pressed Undecided.";
        }

        repaint();
    }
```
public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
}

Sample output from the ButtonDemo applet

As mentioned, in addition to comparing button action command strings, you can also determine which button has been pressed by comparing the object obtained from the `getSource()` method to the button objects that you added to the window. To do this, you must keep a list of the objects when they are added. The following applet shows this approach:
In this version, the program stores each button reference in an array when the buttons are added to the applet.
window. (Recall that the add() method returns a reference to the button when it is added.) Inside actionPerformed(), this array is then used to determine which button has been pressed.

For simple programs, it is usually easier to recognize buttons by their labels. However, in situations in which you will be changing the label inside a button during the execution of your program, or using buttons that have the same label, it may be easier to determine which button has been pushed by using its object reference. It is also possible to set the action command string associated with a button to something other than its label by calling setActionCommand(). This method changes the action command string, but does not affect the string used to label the button. Thus, setting the action command enables the action command and the label of a button to differ.

Applying Check Boxes
A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class.

Checkbox supports these constructors:

Checkbox() throws HeadlessException
Checkbox(String str) throws HeadlessException

Checkbox(String str, boolean on) throws HeadlessException

Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException

Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by str. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null. (Check box groups are described in the next section.) The value of on determines the initial state of the check box.

To retrieve the current state of a check box, call getState( ). To set its state, call setState( ). You can obtain the current label associated with a check box by calling getLabel( ). To set the label, call setLabel( ). These methods are as follows:

boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)

Here, if on is true, the box is checked. If it is false, the box is cleared. The string passed in str becomes the new label associated with the invoking check box.

Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/**<applet code="CheckboxDemo" width=240 height=200>
</applet>*/

public class CheckboxDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox winXP, win7, solaris, mac;

    public void init() {
        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        add(winXP);
        add(win7);
        add(solaris);
        add(mac);
        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
}
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// Display current state of the check boxes.
public void paint(Graphics g) {
    msg = "Current state: ";
g.drawString(msg, 6, 80);
    msg = " Windows XP: " + winXP.getState();
g.drawString(msg, 6, 100);
    msg = " Windows 7: " + win7.getState();
g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
    msg = " Mac OS: " + mac.getState();
g.drawString(msg, 6, 160);
}

Sample output is shown in Figure 25-2.

Figure 25-2 Sample output from the CheckboxDemo applet
It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type CheckboxGroup. Only the default constructor is defined, which creates an empty group.

You can determine which check box in a group is currently selected by calling `getSelectedCheckbox()` . You can set a check box by calling `setSelectedCheckbox()` . These methods are as follows:

```java
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

Here, which is the check box that you want to be selected. The previously selected check box will be turned off.

Here is a program that uses check boxes that are part of a group:
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*  
   <applet code="CBGroup" width=240 height=200>
</applet>
*/

public class CBGroup extends Applet implements ItemListener {
    String msg = ""
    Checkbox winXP, win7, solaris, mac;
    CheckboxGroup cbg;

    public void init() {
        cbg = new CheckboxGroup();
        winXP = new Checkbox("Windows XP", cbg, true);
        win7 = new Checkbox("Windows 7", cbg, false);
        solaris = new Checkbox("Solaris", cbg, false);
        mac = new Checkbox("Mac OS", cbg, false);

        add(winXP);
        add(win7);
        add(solaris);
        add(mac);
        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current selection: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
    }
}

Output generated by the CBGroup applet is shown in
Choice Controls

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. When inactive, a Choice component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice defines only the default constructor, which creates an empty list.

Figure 25-3 Sample output from the CBGroup applet

To add a selection to the list, call add( ). It has this
general form:

    void add(String name)

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur.

To determine which item is currently selected, you may call either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

    String getSelectedItem( )
    int getSelectedIndex( )

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

    int getItemCount( )
    void select(int index)
    void select(String name)

Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which
String getItem(int index)

Here, index specifies the index of the desired item.

Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method.

Here is an example that creates two Choice menus. One selects the operating system. The other selects the browser.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

-->
  <applet code="ChoiceDemo" width=300 height=180>
  </applet>

*/

public class ChoiceDemo extends Applet implements ItemListener {
  Choice os, browser;
  String msg = "";

  public void init() {
    os = new Choice();
    browser = new Choice();

    // add items to os list
    os.add("Windows XP");
    os.add("Windows 7");
    os.add("Solaris");
    os.add("Mac OS");

    // add items to browser list
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Opera");

    // add choice lists to window
    add(os);
    add(browser);

    // register to receive item events
    os.addItemListener(this);
    browser.addItemListener(this);
  }
}
Using Lists

The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices.
in the visible window. It can also be created to allow multiple selections. List provides these constructors:

```java
List() throws HeadlessException
List(int numRows) throws HeadlessException
List(int numRows, boolean multipleSelect) throws HeadlessException
```

The first version creates a List control that allows only one item to be selected at any one time. In the second form, the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if multipleSelect is true, then the user may select two or more items at a time. If it is false, then only one item may be selected.

To add a selection to the list, call `add()`. It has the following two forms:

```java
void add(String name)
void add(String name, int index)
```

Here, `name` is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by `index`. Indexing begins at zero. You can specify −1 to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either `getSelectedItem()` or `getSelectedIndex()`. These
The `getSelectedItem()` method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, `null` is returned. `getSelectedIndex()` returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, −1 is returned.

For lists that allow multiple selection, you must use either `getSelectedItems()` or `getSelectedIndexes()`, shown here, to determine the current selections:

```java
String[] getSelectedItems()
int[] getSelectedIndexes()
```

gSelectedItems() returns an array containing the names of the currently selected items. `getSelectedIndexes()` returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call `getItemCount()`. You can set the currently selected item by using the `select()` method with a zero-based integer index. These methods are shown here:

```java
int getItemCount()
void select(int index)
```
Given an index, you can obtain the name associated with the item at that index by calling `getItem()`, which has this general form:

```java
String getItem(int index)
```

Here, `index` specifies the index of the desired item.

**Handling Lists**

To process list events, you will need to implement the `ActionListener` interface. Each time a List item is double-clicked, an `ActionEvent` object is generated. Its `getActionCommand()` method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an `ItemEvent` object is generated. Its `getStateChange()` method can be used to determine whether a selection or deselection triggered this event. `getItemSelectable()` returns a reference to the object that triggered this event.

Here is an example that converts the `Choice` controls in the preceding section into List components, one multiple choice and the other single choice:
```java
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/**<applet code="ListDemo" width=300 height=180>
</applet>
*/

public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";

    public void init() {
        os = new List(4, true);
        browser = new List(4, false);

        // add items to os list
        os.add("Windows XP");
        os.add("Windows 7");
        os.add("Solaris");
        os.add("Mac OS");

        // add items to browser list
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Opera");

        browser.select(1);

        // add lists to window
        add(os);
        add(browser);

        // register to receive action events
        os.addActionListener(this);
        browser.addActionListener(this);
    }
}```
Sample output generated by the ListDemo applet is shown in Figure 25-5.

Managing Scroll Bars
Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1.
Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the Scrollbar class.

![Figure 25-5 Sample output from the ListDemo applet](image)

Scrollbar defines the following constructors:

Scrollbar() throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max) throws HeadlessException

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If style is Scrollbar.VERTICAL, a vertical scroll bar is created. If style is Scrollbar.HORIZONTAL, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in
initialValue. The number of units represented by the height of the thumb is passed in thumbSize. The minimum and maximum values for the scroll bar are specified by min and max.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using setValues( ), shown here, before it can be used:

```java
void setValues(int initialValue, int thumbSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described.

To obtain the current value of the scroll bar, call getValue( ). It returns the current setting. To set the current value, call setValue( ). These methods are as follows:

```java
int getValue()
void setValue(int newValue)
```

Here, newValue specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

You can also retrieve the minimum and maximum values via getMinimum( ) and getMaximum( ), shown here:

```java
int getMinimum()
int getMaximum()
```
They return the requested quantity.

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling `setUnitIncrement()`. By default, page-up and page-down increments are 10. You can change this value by calling `setBlockIncrement()`. These methods are shown here:

```java
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

Handling Scroll Bars

To process scroll bar events, you need to implement the `AdjustmentListener` interface. Each time a user interacts with a scroll bar, an `AdjustmentEvent` object is generated. Its `getAdjustmentType()` method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK_DECREMENT</td>
<td>A page-down event has been generated.</td>
</tr>
<tr>
<td>BLOCK_INCREMENT</td>
<td>A page-up event has been generated.</td>
</tr>
<tr>
<td>TRACK</td>
<td>An absolute tracking event has been generated.</td>
</tr>
<tr>
<td>UNIT_DECREMENT</td>
<td>The line-down button in a scroll bar has been pressed.</td>
</tr>
<tr>
<td>UNIT_INCREMENT</td>
<td>The line-up button in a scroll bar has been pressed.</td>
</tr>
</tbody>
</table>

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position. Notice the use of `setPreferredSize(`
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
   <applet code="SBDemo" width=300 height=200>
     </applet>
*/

public class SBDemo extends Applet
    implements AdjustmentListener, MouseMotionListener {
    String msg = "";
    Scrollbar vertSB, horzSB;
    public void init() {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        vertSB = new Scrollbar(Scrollbar.VERTICAL,
            0, 1, 0, height);
        vertSB.setPreferredSize(new Dimension(20, 100));

        horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
            0, 1, 0, width);
        horzSB.setPreferredSize(new Dimension(100, 20));

        add(vertSB);
        add(horzSB);

        // register to receive adjustment events
        vertSB.addAdjustmentListener(this);
        horzSB.addAdjustmentListener(this);
        addMouseMotionListener(this);
    }

    public void adjustmentValueChanged(AdjustmentEvent ae) {
        repaint();
    }
}
Sample output from the SBDemo applet is shown in Figure 25-6.
Using a TextField

The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. TextField is a subclass of TextComponent. TextField defines the following constructors:

```java
TextField( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

The first version creates a default text field. The second form creates a text field that is numChars characters wide. The third form initializes the text field with the string contained in str. The fourth form initializes a text field and sets its width.

TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call `getText()`. To set the text, call `setText()`. These methods are as follows:

```java
String getText( )
void setText(String str)
```
Here, $str$ is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using `select()`. Your program can obtain the currently selected text by calling `getSelectedText()`. These methods are shown here:

```java
String getSelectedText()
void select(int startIndex, int endIndex)
```

`getSelectedText()` returns the selected text. The `select()` method selects the characters beginning at `startIndex` and ending at `endIndex − 1`.

You can control whether the contents of a text field may be modified by the user by calling `setEditable()`. You can determine editability by calling `isEditable()`. These methods are shown here:

```java
boolean isEditable()
void setEditable(boolean canEdit)
```

`isEditable()` returns true if the text may be changed and false if not. In `setEditable()`, if `canEdit` is true, the text may be changed. If it is false, the text cannot be altered.

There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling `setEchoChar()`. This method specifies a single character that the `TextField` will display when
characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( ) method. You can retrieve the echo character by calling the getEchoChar( ) method. These methods are as follows:

void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )

Here, ch specifies the character to be echoed. If ch is zero, then normal echoing is restored.

Handling a TextField
Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Here is an example that creates the classic user name and password screen:
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
   <applet code="TextFieldDemo" width=380 height=150>
    </applet>
*/

public class TextFieldDemo extends Applet
    implements ActionListener {

    TextField name, pass;

    public void init() {
        Label namep = new Label("Name: ", Label.RIGHT);
        Label passp = new Label("Password: ", Label.RIGHT);
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');
        add(namep);
        add(name);
        add(passp);
        add(pass);

        // register to receive action events
        name.addActionListener(this);
        pass.addActionListener(this);
    }

    // User pressed Enter.
    public void actionPerformed(ActionEvent ae) {
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString("Name: " + name.getText(6, 60);
        g.drawString("Selected text in name: "
            + name.getSelectedText(6, 80);
        g.drawString("Password: " + pass.getText(6, 100);
    }

Sample output from the TextFieldDemo applet is shown in Figure 25-7.
Using a TextArea
Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea. Following are the constructors for TextArea:

```java
TextArea() throws HeadlessException
TextArea(int numLines, int numChars) throws HeadlessException
TextArea(String str) throws HeadlessException
TextArea(String str, int numLines, int numChars) throws HeadlessException
TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException
```

![Applet Viewer: TextFieldDemo](image)

**Figure 25-7** Sample output from the TextFieldDemo applet

Here, numLines specifies the height, in lines, of the text area, and numChars specifies its width, in characters.
Initial text can be specified by \texttt{str}. In the fifth form, you can specify the scroll bars that you want the control to have. \texttt{sBars} must be one of these values:

<table>
<thead>
<tr>
<th>SCROLLBARS_BOTH</th>
<th>SCROLLBARS_NONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCROLLBARS_HORIZONTAL_ONLY</td>
<td>SCROLLBARS_VERTICAL_ONLY</td>
</tr>
</tbody>
</table>

TextArea is a subclass of TextComponent. Therefore, it supports the \texttt{getText}, \texttt{setText}, \texttt{getSelectedText}, \texttt{select}, \texttt{isEditable}, and \texttt{setEditable} methods described in the preceding section.

TextArea adds the following methods:

\begin{verbatim}
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
\end{verbatim}

The \texttt{append} method appends the string specified by \texttt{str} to the end of the current text. \texttt{insert} inserts the string passed in \texttt{str} at the specified index. To replace text, call \texttt{replaceRange}. It replaces the characters from \texttt{startIndex} to \texttt{endIndex-1}, with the replacement text passed in \texttt{str}.

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Normally, your program simply obtains the current text when it is needed. You can, however, listen for TextEvents, if you choose.

The following program creates a TextArea control:
Here is sample output from the TextAreaDemo applet:
Understanding Layout Managers

All of the components that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. If you have programmed for other GUI environments, such as Windows, then you may have laid out your controls by hand. While it is possible to lay out Java controls by hand, too, you generally won’t want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven’t been realized. This is a
chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manager is set by the setLayout() method. If no call to setLayout() is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The setLayout() method has the following general form:

```java
void setLayout(LayoutManager layoutObj)
```

Here, layoutObj is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for layoutObj. If you do this, you will need to determine the shape and position of each component manually, using the setBounds() method defined by Component. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its minimumLayoutSize() and preferredLayoutSize()
methods. Each component that is being managed by a layout manager contains the `getPreferredSize()` and `getMinimumSize()` methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined `LayoutManager` classes, several of which are described next. You can use the layout manager that best fits your application.

**FlowLayout**

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container’s component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for FlowLayout:

```
FlowLayout()
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```
The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for how are as follows:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Here is a version of the CheckboxDemo applet shown earlier in this chapter, modified so that it uses left-aligned flow layout:
import java.awt.*;
import java.awt.event.*;
import java.applet."

/*
   <applet code="FlowLayoutDemo" width=240 height=200>
   </applet>
*/

public class FlowLayoutDemo extends Applet
    implements ItemListener {

    String msg = "";
    Checkbox winXP, win7, solaris, mac;
    public void init() {
        // set left-aligned flow layout
        setLayout(new FlowLayout(FlowLayout.LEFT));

        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        add(winXP);
        add(win7);
        add(solaris);
        add(mac);

        // register to receive item events
        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
}
Here is sample output generated by the `FlowLayoutDemo` applet. Compare this with the output from the `CheckboxDemo` applet, shown earlier in Figure 25-2.

![Applet Viewer: FlowL...](image)

- Windows XP  ✔
- Windows 7  ✔
- Solaris  
- Mac OS  

Current state:
- Windows XP: true
- Windows 7: true
- Solaris: false
- Mac: false

Applet started.
The `BorderLayout` class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by `BorderLayout`:

```java
BorderLayout( )
BorderLayout(int horz, int vert)
```

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in `horz` and `vert`, respectively.

`BorderLayout` defines the following constants that specify the regions:

<table>
<thead>
<tr>
<th><code>BorderLayout.CENTER</code></th>
<th><code>BorderLayout.SOUTH</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BorderLayout.EAST</code></td>
<td><code>BorderLayout.WEST</code></td>
</tr>
<tr>
<td><code>BorderLayout.NORTH</code></td>
<td></td>
</tr>
</tbody>
</table>

When adding components, you will use these constants with the following form of `add( )`, which is defined by `Container`:

```java
void add(Component compObj, Object region)
```

Here, `compObj` is the component to be added, and `region` specifies where the component will be added.

Here is an example of a `BorderLayout` with a component in each layout area:
public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());

        add(new Button("This is across the top."), BorderLayout.NORTH);
        add(new Label("The footer message might go here."), BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " +
                      "himself to the world;\n" +
                      "the unreasonable one persists in " +
                      "trying to adapt the world to himself.\n" +
                      "Therefore all progress depends " +
                      "on the unreasonable man.\n" +
                      "       - George Bernard Shaw\n"
            ;

        add(new TextArea(msg), BorderLayout.CENTER);
    }
}

Sample output from the BorderLayoutDemo applet is shown here:
Using Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the `getInsets()` method that is defined by `Container`. This method returns an `Insets` object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for `Insets` is shown here:

```
Insets(int top, int left, int bottom, int right)
```

The values passed in `top`, `left`, `bottom`, and `right` specify the amount of space between the container and its enclosing window.

The `getInsets()` method has this general form:
Insets getInsets( )

When overriding this method, you must return a new Insets object that contains the inset spacing you desire.

Here is the preceding BorderLayout example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.
import java.awt.*;
import java.applet.*;
import java.util.*;

/**
 <applet code="InsetsDemo" width=400 height=200>
 </applet>
 */

public class InsetsDemo extends Applet {
    public void init() {
        // set background color so insets can be easily seen
        setBackground(Color.cyan);

        setLayout(new BorderLayout());
        add(new Button("This is across the top."), BorderLayout.NORTH);
        add(new Label("The footer message might go here."), BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " + 
            "himself to the world;\n" + 
            "the unreasonable one persists in " + 
            "trying to adapt the world to himself.\n" + 
            "Therefore all progress depends " + 
            "on the unreasonable man.\n\n" + 
            " - George Bernard Shaw\n\n";

        add(new TextArea(msg), BorderLayout.CENTER);
    }

    // add insets
    public Insets getInsets() {
        return new Insets(10, 10, 10, 10);
    }
}

Output from the InsetsDemo applet is shown here:
GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns. The constructors supported by GridLayout are shown here:

- `GridLayout()`
- `GridLayout(int numRows, int numColumns)`
- `GridLayout(int numRows, int numColumns, int horz, int vert)`

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in `horz` and `vert`, respectively. Either `numRows` or `numColumns` can be zero. Specifying `numRows` as zero allows for unlimited-length columns.
Specifying numColumns as zero allows for unlimited-length rows.

Here is a sample program that creates a $4 \times 4$ grid and fills it in with 15 buttons, each labeled with its index:

```java
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/

public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                    addButton("" + k);
            }
        }
    }
}
```

Following is the output generated by the GridLayoutDemo applet:
TIP

You might try using this example as the starting point for a 15-square puzzle.

CardLayout

The CardLayout class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:

CardLayout( )
CardLayout(int horz, int vert)

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type Panel. This panel must have CardLayout selected as its layout manager. The cards that form the deck are also typically objects of type Panel. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which CardLayout is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of add() when adding cards to a panel:

```java
void add(Component panelObj, Object name)
```

Here, name is a string that specifies the name of the card whose panel is specified by panelObj.

After you have created a deck, your program activates
a card by calling one of the following methods defined by CardLayout:

    void first(Container deck)
    void last(Container deck)
    void next(Container deck)
    void previous(Container deck)
    void show(Container deck, String cardName)

Here, deck is a reference to the container (usually a panel) that holds the cards, and cardName is the name of a card. Calling first() causes the first card in the deck to be shown. To show the last card, call last(). To show the next card, call next(). To show the previous card, call previous(). Both next() and previous() automatically cycle back to the top or bottom of the deck, respectively. The show() method displays the card whose name is passed in cardName.

The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Mac OS and Solaris are displayed in the other card.
import java.awt.*;
import java.awt.event.*;
import java.applet. *;

/*
 <applet code="CardLayoutDemo" width=300 height=100>
 </applet>
*/
public class CardLayoutDemo extends Applet
    implements ActionListener, MouseListener {

    Checkbox winXP, win7, solaris, mac;
    Panel osCards;
    CardLayout cardLO;
    Button Win, Other;

    public void init() {
        Win = new Button("Windows");
        Other = new Button("Other");
        add(Win);
        add(Other);

        cardLO = new CardLayout();
        osCards = new Panel();
        osCards.setLayout(cardLO); // set panel layout to card layout

        winXP = new Checkbox("Windows XP", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        // add Windows check boxes to a panel
        Panel winPan = new Panel();
        winPan.add(winXP);
        winPan.add(win7);
    }
// add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(solaris);
otherPan.add(mac);

// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");

// add cards to main applet panel
add(osCards);

// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);

// register mouse events
addMouseListener(this);

// Cycle through panels.
public void mousePressed(MouseEvent me) {
    cardLO.next(osCards);
}

// Provide empty implementations for the other MouseListener methods.
public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) {
}
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}

public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {
        cardLO.show(osCards, "Windows");
    } else {
        cardLO.show(osCards, "Other");
    }
}
applet. Each card is activated by pushing its button. You can also cycle through the cards by clicking the mouse.

GridBagLayout

Although the preceding layouts are perfectly acceptable for many uses, some situations will require that you take a bit more control over how the components are arranged. A good way to do this is to use a grid bag layout, which is specified by the GridBagLayout class. What makes the grid bag useful is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of
columns. This is why the layout is called a grid bag. It's a collection of small grids joined together.

The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type GridBagConstraints. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

The general procedure for using a grid bag is to first create a new GridBagLayout object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager. Although GridBagLayout is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works.

GridBagLayout defines only one constructor, which is shown here:

```java
GridBagLayout()
```

GridBagLayout defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: setConstraints( ). It is shown here:

```java
void setConstraints(Component comp,
                    GridBagConstraints cons)
```

Here, comp is the component for which the constraints
specified by cons apply. This method sets the constraints that apply to each component in the grid bag.

The key to successfully using GridBagLayout is the proper setting of the constraints, which are stored in a GridBagConstraints object. GridBagConstraints defines several fields that you can set to govern the size, placement, and spacing of a component. These are shown in Table 25-1. Several are described in greater detail in the following discussion.

GridBagConstraints also defines several static fields that contain standard constraint values, such as GridBagConstraints.CENTER and GridBagConstraints.VERTICAL.
<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>int anchor</td>
<td>Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER.</td>
</tr>
<tr>
<td>int fill</td>
<td>Specifies how a component is resized if the component is smaller than its cell. Valid values are GridBagConstraints.NONE (the default), GridBagConstraints.HORIZONTAL, GridBagConstraints.VERTICAL, GridBagConstraints.BOTH.</td>
</tr>
<tr>
<td>int gridheight</td>
<td>Specifies the height of component in terms of cells. The default is 1.</td>
</tr>
<tr>
<td>int gridwidth</td>
<td>Specifies the width of component in terms of cells. The default is 1.</td>
</tr>
<tr>
<td>int gridx</td>
<td>Specifies the X coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE.</td>
</tr>
<tr>
<td>int gridy</td>
<td>Specifies the Y coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE.</td>
</tr>
<tr>
<td>Insets insets</td>
<td>Specifies the insets. Default insets are all zero.</td>
</tr>
<tr>
<td>int ipadx</td>
<td>Specifies extra horizontal space that surrounds a component within a cell. The default is 0.</td>
</tr>
<tr>
<td>int ipady</td>
<td>Specifies extra vertical space that surrounds a component within a cell. The default is 0.</td>
</tr>
<tr>
<td>double weightx</td>
<td>Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.</td>
</tr>
<tr>
<td>double weighty</td>
<td>Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window.</td>
</tr>
</tbody>
</table>

Table 25-1 Constraint Fields Defined by GridBagConstraints

When a component is smaller than its cell, you can use the anchor field to specify where within the cell the component’s top-left corner will be located. There are three types of values that you can give to anchor. The first are absolute:
As their names imply, these values cause the component to be placed at the specific locations.

The second type of values that can be given to anchor is relative, which means the values are relative to the container’s orientation, which might differ for non-Western languages. The relative values are shown here:

<table>
<thead>
<tr>
<th>GridBagConstraints.FIRST_LINE_END</th>
<th>GridBagConstraints.LINE_END</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridBagConstraints.FIRST_LINE_START</td>
<td>GridBagConstraints.LINE_START</td>
</tr>
<tr>
<td>GridBagConstraints.LAST_LINE_END</td>
<td>GridBagConstraints.PAGE_END</td>
</tr>
<tr>
<td>GridBagConstraints.LAST_LINE_START</td>
<td>GridBagConstraints.PAGE_START</td>
</tr>
</tbody>
</table>

Their names describe the placement.

The third type of values that can be given to anchor allows you to position components relative to the baseline of the row. These values are shown here:

<table>
<thead>
<tr>
<th>GridBagConstraints.BASELINE</th>
<th>GridBagConstraints.BASELINE_LEADING</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridBagConstraints.BASELINE_TRAILING</td>
<td>GridBagConstraints.ABOVE_BASELINE</td>
</tr>
<tr>
<td>GridBagConstraints.ABOVE_BASELINE_LEADING</td>
<td>GridBagConstraints.ABOVE_BASELINE_TRAILING</td>
</tr>
<tr>
<td>GridBagConstraints.BELOW_BASELINE</td>
<td>GridBagConstraints.BELOW_BASELINE_LEADING</td>
</tr>
<tr>
<td>GridBagConstraints.BELOW_BASELINE_TRAILING</td>
<td></td>
</tr>
</tbody>
</table>

The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

The weightx and weighty fields are both quite
important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column’s allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.

The gridwidth variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use GridBagConstraints.REMAINDER. To specify that a component use the next-to-last cell in a row, use GridBagConstraints.RELATIVE. The gridheight constraint works the same way, but in the vertical direction.

You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to ipadx. To pad vertically, assign a value to ipady.

Here is an example that uses GridBagLayout to demonstrate several of the points just discussed:
public class GridBagDemo extends Applet
    implements ItemListener {

    String msg = "";
    Checkbox winXP, win7, solaris, mac;

    public void init() {
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);

        // Define check boxes.
        winXP = new Checkbox("Windows XP ", null, true);
        win7 = new Checkbox("Windows 7");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        // Define the grid bag.
/ Use default row weight of 0 for first row.
gbc.weightx = 1.0; // use a column weight of 1

  gbc.ipadx = 200; // pad by 200 units
  gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left

  gbc.anchor = GridBagConstraints.NORTHEAST;

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(winXP, gbc);

  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(win7, gbc);

  // Give second row a weight of 1.
  gbc.weighty = 1.0;

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(solaris, gbc);

  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(mac, gbc);
The output produced by the program is shown here.
In this layout, the operating system check boxes are positioned in a $2 \times 2$ grid. Each cell has a horizontal padding of 200. Each component is inset slightly (by 4 units) from the top left. The column weight is set to 1, which causes any extra horizontal space to be distributed evenly between the columns. The first row uses a default weight of 0; the second has a weight of 1. This means that any extra vertical space is added to the second row.

GridBagLayout is a powerful layout manager. It is worth taking some time to experiment with and explore. Once you understand what the various settings do, you can use GridBagLayout to position components with a high degree of precision.

Menu Bars and Menus
A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This
concept is implemented in the AWT by the following classes: MenuBar, Menu, and MenuItem. In general, a menu bar contains one or more Menu objects. Each Menu object contains a list of MenuItem objects. Each MenuItem object represents something that can be selected by the user. Since Menu is a subclass of MenuItem, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type CheckboxMenuItem and will have a check mark next to them when they are selected.

To create a menu bar, first create an instance of MenuBar. This class defines only the default constructor. Next, create instances of Menu that will define the selections displayed on the bar. Following are the constructors for Menu:

```
Menu() throws HeadlessException
Menu(String optionName) throws HeadlessException
Menu(String optionName, boolean removable) throws HeadlessException
```

Here, optionName specifies the name of the menu selection. If removable is true, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu.

Individual menu items are of type MenuItem. It
defines these constructors:

```java
MenuItem( ) throws HeadlessException
MenuItem(String itemName) throws HeadlessException
MenuItem(String itemName, MenuShortcut keyAccel)
    throws HeadlessException
```

Here, `itemName` is the name shown in the menu, and `keyAccel` is the menu shortcut for this item.

You can disable or enable a menu item by using the `setEnabled( )` method. Its form is shown here:

```java
void setEnabled(boolean enabledFlag)
```

If the argument `enabledFlag` is `true`, the menu item is enabled. If `false`, the menu item is disabled.

You can determine an item’s status by calling `isEnabled( )`. This method is shown here:

```java
boolean isEnabled( )
```

`isEnabled( )` returns `true` if the menu item on which it is called is enabled. Otherwise, it returns `false`.

You can change the name of a menu item by calling `setLabel( )`. You can retrieve the current name by using `getLabel( )`. These methods are as follows:

```java
void setLabel(String newName)
String getLabel( )
```
Here, newName becomes the new name of the invoking menu item. getLabel( ) returns the current name.

You can create a checkable menu item by using a subclass of MenuItem called CheckboxMenuItem. It has these constructors:

CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String itemName) throws HeadlessException
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException

Here, itemName is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if on is true, the checkable entry is initially checked. Otherwise, it is cleared.

You can obtain the status of a checkable item by calling getState( ). You can set it to a known state by using setState( ). These methods are shown here:

boolean getState( )
void setState(boolean checked)

If the item is checked, getState( ) returns true. Otherwise, it returns false. To check an item, pass true to setState( ). To clear an item, pass false.

Once you have created a menu item, you must add the
item to a Menu object by using add( ), which has the following general form:

    MenuItem add(MenuItem item)

Here, item is the item being added. Items are added to a menu in the order in which the calls to add( ) take place. The item is returned.

Once you have added all items to a Menu object, you can add that object to the menu bar by using this version of add( ) defined by MenuBar:

    Menu add(Menu menu)

Here, menu is the menu being added. The menu is returned.

Menus generate events only when an item of type MenuItem or CheckboxMenuItem is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an ActionEvent object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling setActionCommand( ) on the menu item. Each time a check box menu item is checked or unchecked, an ItemEvent object is generated. Thus, you must implement the ActionListener and/or ItemListener interfaces in order to handle these menu events.
The `getItem()` method of `ItemEvent` returns a reference to the item that generated this event. The general form of this method is shown here:

```
Object getItem()
```

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```java
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
   <applet code="MenuDemo" width=250 height=250>
   </applet>
*/

// Create a subclass of Frame.
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;
    
    MenuFrame(String title) {
        super(title);

        // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // create the menu items
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4, item5;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(item4 = new MenuItem("-"));
        file.add(item5 = new MenuItem("Quit..."));
        mbar.add(file);
    }
}
```
Menu edit = new Menu("Edit");
MenuItem item6, item7, item8, item9;
edit.add(item6 = new MenuItem("Cut"));
edit.add(item7 = new MenuItem("Copy"));
edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));

Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);

// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);

mbar.add(edit);
// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
dbg.addItemListener(handler);
test.addItemListener(handler);

// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
        g.drawString("Debug is on.", 10, 220);
    else
        g.drawString("Debug is off.", 10, 220);

    if(test.getState())
        g.drawString("Testing is on.", 10, 240);
    else
        g.drawString("Testing is off.", 10, 240);
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;

    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;

    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
public void actionPerformed(ActionEvent ae) {
    String msg = "You selected ";
    String arg = ae.getActionCommand();
    if(arg.equals("New..."))
        msg += "New."
    else if(arg.equals("Open..."))
        msg += "Open."
    else if(arg.equals("Close"))
        msg += "Close."
    else if(arg.equals("Quit..."))
        msg += "Quit."
    else if(arg.equals("Edit"))
        msg += "Edit."
    else if(arg.equals("Cut"))
        msg += "Cut."
    else if(arg.equals("Copy"))
        msg += "Copy."
    else if(arg.equals("Paste"))
        msg += "Paste."
    else if(arg.equals("First"))
        msg += "First."
    else if(arg.equals("Second"))
        msg += "Second."
    else if(arg.equals("Third"))
        msg += "Third."
    else if(arg.equals("Debug"))
        msg += "Debug."
    else if(arg.equals("Testing"))
        msg += "Testing."
    menuFrame.msg = msg;
    menuFrame.repaint();
}
Sample output from the MenuDemo applet is shown in Figure 25-8.
There is one other menu-related class that you might find interesting: PopupMenu. It works just like Menu, but produces a menu that can be displayed at a specific location. PopupMenu provides a flexible, useful alternative for some types of menuing situations.

Dialog Boxes
Often, you will want to use a dialog box to hold a set of related controls. Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don’t have menu bars, but in other respects, they function like frame windows. (You can add controls to them, for example, in the same way that you add controls to a frame window.) Dialog boxes may be modal or modeless. When a modal dialog box is active, all input is directed to it until it is closed. This
means that you cannot access other parts of your 
program until you have closed the dialog box. When a 
modeless dialog box is active, input focus can be directed to 
another window in your program. Thus, other parts of 
your program remain active and accessible. Dialog boxes 
are of type Dialog. Two commonly used constructors are 
shown here:

```java
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

Here, `parentWindow` is the owner of the dialog box. If 
`mode` is true, the dialog box is modal. Otherwise, it is 
modeless. The title of the dialog box can be passed in 
title. Generally, you will subclass `Dialog`, adding the 
functionality required by your application.

Following is a modified version of the preceding menu program that displays a modeless dialog box when the New option is chosen. Notice that when the dialog box is closed, `dispose()` is called. This method is defined by `Window`, and it frees all system resources associated with the dialog box window.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/
   <applet code=DialogDemo width=250 height=250>
   
   </applet>
/*

// Create a subclass of Dialog.

class SampleDialog extends Dialog implements ActionListener {
    SampleDialog(Frame parent, String title) {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(300, 200);

        add(new Label("Press this button:"));
        Button b;
        add(b = new Button("Cancel"));
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        dispose();
    }

    public void paint(Graphics g) {
        g.drawString("This is in the dialog box", 10, 70);
    }
}
// Create a subclass of Frame.
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // create the menu items
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4;
        file.add(item1 = new MenuItem("New...")
        file.add(item2 = new MenuItem("Open...")
        file.add(item3 = new MenuItem("Close")
        file.add(new MenuItem("-")
        file.add(item4 = new MenuItem("Quit...")
        mbar.add(file);

        Menu edit = new Menu("Edit");
        MenuItem item5, item6, item7;
        edit.add(item5 = new MenuItem("Cut")
        edit.add(item6 = new MenuItem("Copy")
        edit.add(item7 = new MenuItem("Paste")
        edit.add(new MenuItem("-")

        Menu sub = new Menu("Special", true);
        MenuItem item8, item9, item10;
        sub.add(item8 = new MenuItem("First")
        sub.add(item9 = new MenuItem("Second")
        sub.add(item10 = new MenuItem("Third")
        edit.add(sub);
// these are checkable menu items
decode = new CheckboxMenuitem("Debug");
decode.add(debug);
test = new CheckboxMenuitem("Testing");
test.add(test);

mbar.add(edit);

// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
decode.addActionListener(handler);
test.addActionListener(handler);

// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);

// register it to receive those events
addWindowListener(adapter);
public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
        g.drawString("Debug is on.", 10, 220);
    else
        g.drawString("Debug is off.", 10, 220);

    if(test.getState())
        g.drawString("Testing is on.", 10, 240);
    else
        g.drawString("Testing is off.", 10, 240);
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;

    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.dispose();
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
// Handle action events.
public void actionPerformed(ActionEvent ae) {
    String msg = "You selected ";
    String arg = ae.getActionCommand();
    // Activate a dialog box when New is selected.
    if(arg.equals("New...")) {
        msg += "New.";
        SampleDialog d = new
            SampleDialog(menuFrame, "New Dialog Box");
        d.setVisible(true);
    }
    // Try defining other dialog boxes for these options.
    else if(arg.equals("Open..."))
        msg += "Open.";
    else if(arg.equals("Close"))
        msg += "Close.";
    else if(arg.equals("Quit..."))
        msg += "Quit.";
    else if(arg.equals("Edit"))
        msg += "Edit.";
    else if(arg.equals("Cut"))
        msg += "Cut.";
    else if(arg.equals("Copy"))
        msg += "Copy.";
    else if(arg.equals("Paste"))
        msg += "Paste.";
    else if(arg.equals("First"))
        msg += "First.";
    else if(arg.equals("Second"))
        msg += "Second.";
    else if(arg.equals("Third"))
        msg += "Third.";
    else if(arg.equals("Debug"))
        msg += "Debug.";
    else if(arg.equals("Testing"))
        msg += "Testing.";
    menuFrame.msg = msg;
    menuFrame.repaint();
}

public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
}
// Create frame window.
public class DialogDemo extends Applet {
    Frame f;
    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(width, height);

        f.setSize(width, height);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }
}

Here is sample output from the DialogDemo applet:
TIP

On your own, try defining dialog boxes for the other options presented by the menus.

FileDialog
Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type FileDialog. This causes a file dialog box to be displayed.

Usually, this is the standard file dialog box provided by the operating system. Here are three FileDialog constructors:
FileDialog(Frame parent)
FileDialog(Frame parent, String boxName)
FileDialog(Frame parent, String boxName, int how)

Here, parent is the owner of the dialog box. The boxName parameter specifies the name displayed in the box’s title bar. If boxName is omitted, the title of the dialog box is empty. If how is FileDialog.LOAD, then the box is selecting a file for reading. If how is FileDialog.SAVE, the box is selecting a file for writing. If how is omitted, the box is selecting a file for reading.

FileDialog provides methods that allow you to determine the name of the file and its path as selected by the user. Here are two examples:

String getDirectory( )
String getFile( )

These methods return the directory and the filename, respectively.

The following program activates the standard file dialog box:
The output generated by this program is shown here.
(The precise configuration of the dialog box may vary.)
One last point: Beginning with JDK 7, you can use `FileDialog` to select a list of files. This functionality is supported by the `setMultipleMode()`, `isMultipleMode()`, and `getFiles()` methods.

Handling Events by Extending AWT Components

The delegation event model was introduced in Chapter 23, and all of the programs in this book so far have used that design. But Java also allows you to handle events by subclassing AWT components. Doing so allows you to handle events in much the same way as they were handled under the original 1.0 version of Java. Of course, this technique is discouraged, because it has the
same disadvantages of the Java 1.0 event model, the
main one being inefficiency. Handling events by
extending AWT components is described in this section
for completeness. However, this technique is not used in
any other sections of this book.

When extending an AWT component, you must call
the enableEvents( ) method of Component. Its general
form is shown here:

protected final void enableEvents(long eventMask)

The eventMask argument is a bit mask that defines the
events to be delivered to this component. The AWTEvent
class defines int constants for making this mask. Several
are shown here:

<table>
<thead>
<tr>
<th>ACTION_EVENT_MASK</th>
<th>KEY_EVENT_MASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADJUSTMENT_EVENT_MASK</td>
<td>MOUSE_EVENT_MASK</td>
</tr>
<tr>
<td>COMPONENT_EVENT_MASK</td>
<td>MOUSE_MOTION_EVENT_MASK</td>
</tr>
<tr>
<td>CONTAINER_EVENT_MASK</td>
<td>MOUSE_WHEEL_EVENT_MASK</td>
</tr>
<tr>
<td>FOCUS_EVENT_MASK</td>
<td>TEXT_EVENT_MASK</td>
</tr>
<tr>
<td>INPUT_METHOD_EVENT_MASK</td>
<td>WINDOW_EVENT_MASK</td>
</tr>
<tr>
<td>ITEM_EVENT_MASK</td>
<td></td>
</tr>
</tbody>
</table>

You must also override the appropriate method from
one of your superclasses in order to process the event. Be
sure to also call the superclass version of the method.
Table 25-2 lists several commonly used methods and the
classes that provide them.

The following sections provide simple programs that
show how to extend several AWT components.
Extending Button

The following program creates an applet that displays a button labeled “Test Button”. When the button is pressed, the string “action event:” is displayed on the status line of the applet viewer or browser, followed by a count of the number of button presses.

The program has one top-level class named ButtonDemo2 that extends Applet. A static integer variable named i is defined and initialized to zero. This records the number of button pushes. The init( ) method instantiates MyButton and adds it to the applet.

MyButton is an inner class that extends Button. Its constructor uses super to pass the label of the button to the superclass constructor. It calls enableEvents( ) so that action events may be received by this object. When an action event is generated, processActionEvent( ) is called. That method displays a string on the status line and calls processActionEvent( ) for the superclass. Because MyButton is an inner class, it has direct access to the showStatus( ) method of ButtonDemo2.
Table 25-2 Commonly Used Event Processing Methods

<table>
<thead>
<tr>
<th>Class</th>
<th>Processing Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>processActionEvent()</td>
</tr>
<tr>
<td>Checkbox</td>
<td>processItemEvent()</td>
</tr>
<tr>
<td>CheckboxMenuItem</td>
<td>processItemEvent()</td>
</tr>
<tr>
<td>Choice</td>
<td>processComponentEvent(), processFocusEvent(), processKeyEvent(), processMouseEvent(), processMouseMotionEvent(), processMouseWheelEvent()</td>
</tr>
<tr>
<td>Component</td>
<td>processActionEvent(), processItemEvent()</td>
</tr>
<tr>
<td>List</td>
<td>processActionEvent(), processItemEvent()</td>
</tr>
<tr>
<td>MenuItem</td>
<td>processActionEvent()</td>
</tr>
<tr>
<td>Scrollbar</td>
<td>processAdjustmentEvent()</td>
</tr>
<tr>
<td>TextComponent</td>
<td>processTextEvent()</td>
</tr>
</tbody>
</table>

/*
* <applet code=ButtonDemo2 width=200 height=100>
* </applet>
*/

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ButtonDemo2 extends Applet {  
  MyButton myButton;
  static int i = 0;
  public void init() { 
    myButton = new MyButton("Test Button");
    add(myButton);
  }
  class MyButton extends Button {  
    public MyButton(String label) { 
      super(label);
      enableEvents(AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent ae) { 
      showStatus("action event: " + i++);
      super.processActionEvent(ae);
    }
  }
}

Extending Checkbox
The following program creates an applet that displays three check boxes labeled “Item 1”, “Item 2”, and “Item 3”. When a check box is selected or deselected, a string containing the name and state of that check box is displayed on the status line of the applet viewer or browser.

The program has one top-level class named CheckboxDemo2 that extends Applet. Its init( ) method creates three instances of MyCheckbox and adds these to the applet. MyCheckbox is an inner class that extends Checkbox. Its constructor uses super to pass the label of the check box to the superclass constructor. It calls enableEvents( ) so that item events may be received by this object. When an item event is generated, processItemEvent( ) is called. That method displays a string on the status line and calls processItemEvent( ) for the superclass.
The following program reworks the preceding check box example so that the check boxes form a check box group. Thus, only one of the check boxes may be selected at any time.
Extending Choice

The following program creates an applet that displays a choice list with items labeled “Red”, “Green”, and “Blue”. When an entry is selected, a string that contains the name of the color is displayed on the status line of the applet viewer or browser.

There is one top-level class named ChoiceDemo2 that
extends Applet. Its init( ) method creates a choice element and adds it to the applet. MyChoice is an inner class that extends Choice. It calls enableEvents( ) so that item events may be received by this object. When an item event is generated, processItemEvent( ) is called. That method displays a string on the status line and calls processItemEvent( ) for the superclass.

```java
/*
 * <applet code=ChoiceDemo2 width=300 height=100>
 * </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ChoiceDemo2 extends Applet {
    MyChoice choice;
    public void init() {
        choice = new MyChoice();
        choice.add("Red");
        choice.add("Green");
        choice.add("Blue");
        add(choice);
    }
    class MyChoice extends Choice {
        public MyChoice() {
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
        protected void processItemEvent(ItemEvent ie) {
            showStatus("Choice selection: " + getSelectedItem());
            super.processItemEvent(ie);
        }
    }
}

Extending List
The following program modifies the preceding example so that it uses a list instead of a choice menu. There is
one top-level class named ListDemo2 that extends Applet. Its init() method creates a list element and adds it to the applet. MyList is an inner class that extends List. It calls enableEvents() so that both action and item events may be received by this object. When an entry is selected or deselected, processItemEvent() is called. When an entry is double-clicked, processActionEvent() is also called. Both methods display a string and then hand control to the superclass.
Extending Scrollbar

The following program creates an applet that displays a scrollbar. When this control is manipulated, a string is displayed on the status line of the applet viewer or browser. That string includes the value represented by the scrollbar.

There is one top-level class named ScrollbarDemo2 that extends Applet. Its init() method creates a scrollbar:

```java
import java.awt.*;
import java.awt.event.*;
import java.applet.);

public class ListDemo2 extends Applet {  
    MyList list;  
    public void init() {  
        list = new MyList();  
        list.add("Red");  
        list.add("Green");  
        list.add("Blue");  
        add(list);  
    }
    
    class MyList extends List {  
        public MyList() {  
            enableEvents(AWTEvent.ITEM_EVENT_MASK |  
                        AWTEvent.ACTION_EVENT_MASK);

        }
        
        protected void processActionEvent(ActionEvent ae) {  
            showStatus("Action event: " + ae.getActionCommand());  
            super.processActionEvent(ae);
        }

        protected void processItemEvent(ItemEvent ie) {  
            showStatus("Item event: " + getSelectedItem());  
            super.processItemEvent(ie);
        }
    }
}
```
element and adds it to the applet. MyScrollbar is an inner class that extends Scrollbar. It calls enableEvents() so that adjustment events may be received by this object. When the scroll bar is manipulated, processAdjustmentEvent() is called. When an entry is selected, processAdjustmentEvent() is called. It displays a string and then hands control to the superclass.

```java
/*
 * <applet code=ScrollbarDemo2 width=300 height=100>
 * </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ScrollbarDemo2 extends Applet {
    MyScrollbar myScrollbar;
    public void init() {
        myScrollbar = new MyScrollbar(Scrollbar.HORIZONTAL,
                                         0, 1, 0, 100);
        myScrollbar.setPreferredSize(new Dimension(100, 20));
        add(myScrollbar);
    }

class MyScrollbar extends Scrollbar {
    public MyScrollbar(int style, int initial, int thumb,
                         int min, int max) {
        super(style, initial, thumb, min, max);
        enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
    }
    protected void processAdjustmentEvent(AdjustmentEvent ae) {
        showStatus("Adjustment event: " + ae.getValue());
        setValue(getValue());
    }
}

A Word About Overriding paint()

Before concluding our examination of AWT controls, a
short word about overriding paint( ) is in order. Although not relevant to the simple AWT examples shown in this book, when overriding paint( ), there are times when it is necessary to call the superclass implementation of paint( ). Therefore, for some programs, you will need to use this paint( ) skeleton:

```java
public void paint(Graphics g) {

    // code to repaint this window

    // Call superclass paint()
    super.paint(g);
}
```

In Java, there are two general types of components: heavyweight and lightweight. A heavyweight component has its own native window, which is called its peer. A lightweight component is implemented completely in Java code and uses the window provided by an ancestor. The AWT controls described and used in this chapter are all heavyweight. However, if a container holds any lightweight components (that is, has lightweight child components), your override of paint( ) for that container must call super.paint( ). By calling super.paint( ), you ensure that any lightweight child components, such as lightweight controls, get properly painted. If you are unsure of a child component’s type, you can call isLightweight( ), defined by Component, to find out. It returns true if the component is lightweight, and false otherwise.
CHAPTER 26

Images

This chapter examines the AWT’s Image class and the java.awt.image package. Together, they provide support for imaging (the display and manipulation of graphical images). An image is simply a rectangular graphical object. Images are a key component of web design. In fact, the inclusion of the `<img>` tag in the Mosaic browser at NCSA (National Center for Supercomputer Applications) is what caused the Web to begin to grow explosively in 1993. This tag was used to include an image inline with the flow of hypertext. Java expands upon this basic concept, allowing images to be managed under program control. Because of its importance, Java provides extensive support for imaging.

Images are objects of the Image class, which is part of the java.awt package. Images are manipulated using the classes found in the java.awt.image package. There are a large number of imaging classes and interfaces defined by java.awt.image, and it is not possible to examine them all. Instead, we will focus on those that form the foundation of imaging. Here are the java.awt.image classes discussed in this chapter:

<table>
<thead>
<tr>
<th>CropImageFilter</th>
<th>MemoryImageSource</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilteredImageSource</td>
<td>PixelGrabber</td>
</tr>
<tr>
<td>ImageFilter</td>
<td>RGBImageFilter</td>
</tr>
</tbody>
</table>

These are the interfaces that we will use:

| ImageConsumer | ImageObserver | ImageProducer |

Also examined is the MediaTracker class, which is part
File Formats

Originally, web images could only be in GIF format. The GIF image format was created by CompuServe in 1987 to make it possible for images to be viewed while online, so it was well suited to the Internet. GIF images can have only up to 256 colors each. This limitation caused the major browser vendors to add support for JPEG images in 1995. The JPEG format was created by a group of photographic experts to store full-color-spectrum, continuous-tone images. These images, when properly created, can be of much higher fidelity as well as more highly compressed than a GIF encoding of the same source image. Another file format is PNG. It too is an alternative to GIF. In almost all cases, you will never care or notice which format is being used in your programs. The Java image classes abstract the differences behind a clean interface.

Image Fundamentals: Creating, Loading, and Displaying

There are three common operations that occur when you work with images: creating an image, loading an image, and displaying an image. In Java, the Image class is used to refer to images in memory and to images that must be loaded from external sources. Thus, Java provides ways for you to create a new image object and ways to load one. It also provides a means by which an image can be displayed. Let’s look at each.
Creating an Image Object

You might expect that you create a memory image using something like the following:

```java
Image test = new Image(200, 100); // Error -- won't work
```

Not so. Because images must eventually be painted on a window to be seen, the Image class doesn’t have enough information about its environment to create the proper data format for the screen. Therefore, the Component class in java.awt has a factory method called `createImage()` that is used to create Image objects. (Remember that all of the AWT components are subclasses of Component, so all support this method.)

The `createImage()` method has the following two forms:

- `Image createImage(ImageProducer imgProd)`
- `Image createImage(int width, int height)`

The first form returns an image produced by `imgProd`, which is an object of a class that implements the `ImageProducer` interface. (We will look at image producers later.) The second form returns a blank (that is, empty) image that has the specified width and height. Here is an example:

```java
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

This creates an instance of Canvas and then calls the `createImage()` method to actually make an Image object. At this point, the image is blank. Later you will see how
Loading an Image

The other way to obtain an image is to load one. One way to do this is to use the getImage( ) method defined by the Applet class. It has the following forms:

- `Image getImage(URL url)`
- `Image getImage(URL url, String imageName)`

The first version returns an Image object that encapsulates the image found at the location specified by url. The second version returns an Image object that encapsulates the image found at the location specified by url and having the name specified by imageName.

Displaying an Image

Once you have an image, you can display it by using `drawImage( )`, which is a member of the Graphics class. It has several forms. The one we will be using is shown here:

- `boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)`

This displays the image passed in imgObj with its upper-left corner specified by left and top. imgOb is a reference to a class that implements the ImageObserver interface. This interface is implemented by all AWT (and Swing) components. An image observer is an object that can monitor an image while it loads. ImageObserver is described in the next section.
With `getImage()` and `drawImage()`, it is actually quite easy to load and display an image. Here is a sample applet that loads and displays a single image. The file `seattle.jpg` is loaded, but you can substitute any GIF, JPG, or PNG file you like (just make sure it is available in the same directory with the HTML file that contains the applet).

```java
/*
 * <applet code="SimpleImageLoad" width=248 height=146>
 * <param name="img" value="seattle.jpg">
 * </applet>
 */
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet {
    Image img;

    public void init() {
        img = getImage(getDocumentBase(), getParameter("img"));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}

In the `init()` method, the `img` variable is assigned to the image returned by `getImage()`. The `getImage()` method uses the string returned by `getParameter("img")` as the filename for the image. This image is loaded from a URL that is relative to the result of `getDocumentBase()`, which is the URL of the HTML page this applet tag was in. The filename returned by `getParameter("img")` comes from the applet tag `<param name="img" value="seattle.jpg">`. This is the equivalent, if a little
slower, of using the HTML tag `<img src="seattle.jpg" width=248 height=146>`. Figure 26-1 shows what it looks like when you run the program.

![Image of Seattle skyline](seattle.jpg)

Figure 26-1 Sample output from SimpleImageLoad

When this applet runs, it starts loading img in the `init()` method. Onscreen you can see the image as it loads from the network, because Applet’s implementation of the `ImageObserver` interface calls `paint()` every time more image data arrives.

Seeing the image load is somewhat informative, but it might be better if you use the time it takes to load the image to do other things in parallel. That way, the fully formed image can simply appear on the screen in an instant, once it is fully loaded. You can use `ImageObserver`, described next, to monitor loading an image while you paint the screen with other information.

**ImageObserver**

`ImageObserver` is an interface used to receive notification as an image is being generated, and it defines only one method: `imageUpdate()`. Using an image observer allows you to perform other actions, such as show a progress indicator or an attract screen, as you are
informed of the progress of the download. This kind of notification is very useful when an image is being loaded over a slow network.

The `imageUpdate()` method has this general form:

```java
boolean imageUpdate(Image imgObj, int flags, int left, int top, int width, int height)
```

Here, `imgObj` is the image being loaded, and `flags` is an integer that communicates the status of the update report. The four integers `left`, `top`, `width`, and `height` represent a rectangle that contains different values depending on the values passed in `flags`. `imageUpdate()` should return `false` if it has completed loading, and `true` if there is more image to process.

The `flags` parameter contains one or more bit flags defined as static variables inside the `ImageObserver` interface. These flags and the information they provide are listed in Table 26-1.

The Applet class has an implementation of the `imageUpdate()` method for the `ImageObserver` interface that is used to repaint images as they are loaded. You can override this method in your class to change that behavior.
Table 26-1 Bit Flags of the `imageUpdate()` flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIDTH</td>
<td>The <em>width</em> parameter is valid and contains the width of the image.</td>
</tr>
<tr>
<td>HEIGHT</td>
<td>The <em>height</em> parameter is valid and contains the height of the image.</td>
</tr>
</tbody>
</table>
| PROPERTIES | The properties associated with the image can now be obtained using `imgObj.getProperty()`.
| SOMEBITS | More pixels needed to draw the image have been received. The parameters *left*, *top*, *width*, and *height* define the rectangle containing the new pixels. |
| FRAMEBITS | A complete frame that is part of a multiframe image, which was previously drawn, has been received. This frame can be displayed. The *left*, *top*, *width*, and *height* parameters are not used. |
| ALLBITS  | The image is now complete. The *left*, *top*, *width*, and *height* parameters are not used. |
| ERROR    | An error has occurred to an image that was being tracked asynchronously. The image is incomplete and cannot be displayed. No further image information will be received. The ABORT flag will also be set to indicate that the image production was aborted. |
| ABORT    | An image that was being tracked asynchronously was aborted before it was complete. However, if an error has not occurred, accessing any part of the image’s data will restart the production of the image. |

---

Here is a simple example of an `imageUpdate()` method:

```java
public boolean imageUpdate(Image img, int flags, int x, int y, int w, int h) {
    if ((flags & ALLBITS) == 0) {
        System.out.println("Still processing the image.");
        return true;
    } else {
        System.out.println("Done processing the image.");
        return false;
    }
}
```

Double Buffering

Not only are images useful for storing pictures, as we’ve just shown, but you can also use them as offscreen drawing surfaces. This allows you to render any image, including text and graphics, to an offscreen buffer that
you can display at a later time. The advantage to doing this is that the image is seen only when it is complete. Drawing a complicated image could take several milliseconds or more, which can be seen by the user as flashing or flickering. This flashing is distracting and causes the user to perceive your rendering as slower than it actually is. Use of an offscreen image to reduce flicker is called double buffering, because the screen is considered a buffer for pixels, and the offscreen image is the second buffer, where you can prepare pixels for display.

Earlier in this chapter, you saw how to create a blank Image object. Now you will see how to draw on that image rather than the screen. As you recall from earlier chapters, you need a Graphics object in order to use any of Java’s rendering methods. Conveniently, the Graphics object that you can use to draw on an Image is available via the getGraphics() method. Here is a code fragment that creates a new image, obtains its graphics context, and fills the entire image with red pixels:

```java
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

Once you have constructed and filled an offscreen image, it will still not be visible. To actually display the image, call drawImage(). Here is an example that draws a time-consuming image, to demonstrate the difference that double buffering can make in perceived drawing
/*
 * <applet code=DoubleBuffer width=250 height=250>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DoubleBuffer extends Applet {
    int gap = 3;
    int mx, my;
    boolean flicker = true;
    Image buffer = null;
    int w, h;

    public void init() {
        Dimension d = getSize();
        w = d.width;
        h = d.height;
        buffer = createImage(w, h);
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent me) {
                mx = me.getX();
                my = me.getY();
                flicker = false;
                repaint();
            }
            public void mouseMoved(MouseEvent me) {
                mx = me.getX();
                my = me.getY();
                flicker = true;
                repaint();
            }
        });
    }
}
This simple applet has a complicated `paint()` method. It fills the background with blue and then draws a red moiré pattern on top of that. It paints some black text on top of that and then paints a yellow circle centered at the coordinates `mx`, `my`. The `mouseMoved()` and `mouseDragged()` methods are overridden to track the mouse position. These methods are identical, except for the setting of the `flicker` Boolean variable. `mouseMoved()` sets `flicker` to true, and `mouseDragged()` sets it to false. This has the effect of calling `repaint()` with `flicker` set to
true when the mouse is moved (but no button is pressed) and set to false when the mouse is dragged with any button pressed.

When paint( ) gets called with flicker set to true, we see each drawing operation as it is executed on the screen. In the case where a mouse button is pressed and paint( ) is called with flicker set to false, we see quite a different picture. The paint( ) method swaps the Graphics reference g with the graphics context that refers to the offscreen canvas, buffer, which we created in init( ). Then all of the drawing operations are invisible. At the end of paint( ), we simply call drawImage( ) to show the results of these drawing methods all at once.

Notice that it is okay to pass in a null as the fourth parameter to drawImage( ). This is the parameter used to pass an ImageObserver object that receives notification of image events. Since this is an image that is not being produced from a network stream, we have no need for notification. The left snapshot in Figure 26-2 is what the applet looks like with the mouse button not pressed. As you can see, the image was in the middle of repainting when this snapshot was taken. The right snapshot shows how, when a mouse button is pressed, the image is always complete and clean due to double buffering.
MediaTracker

A MediaTracker is an object that will check the status of an arbitrary number of images in parallel. To use MediaTracker, you create a new instance and use its addImage( ) method to track the loading status of an image. addImage( ) has the following general forms:

```java
void addImage(Image imgObj, int imgID)
void addImage(Image imgObj, int imgID, int width, int height)
```

Here, imgObj is the image being tracked. Its identification number is passed in imgID. ID numbers do not need to be unique. You can use the same number with several images as a means of identifying them as part of a group. Furthermore, images with lower IDs are given priority over those with higher IDs when loading. In the second form, width and height specify the
Once you’ve registered an image, you can check whether it’s loaded, or you can wait for it to completely load. To check the status of an image, call checkID(). The version used in this chapter is shown here:

```java
boolean checkID(int imgID)
```

Here, `imgID` specifies the ID of the image you want to check. The method returns true if all images that have the specified ID have been loaded (or if an error or user-abort has terminated loading). Otherwise, it returns false. You can use the checkAll() method to see if all images being tracked have been loaded.

You should use MediaTracker when loading a group of images. If all of the images that you’re interested in aren’t downloaded, you can display something else to entertain the user until they all arrive.

**CAUTION**

If you use MediaTracker once you’ve called addImage() on an image, a reference in MediaTracker will prevent the system from garbage collecting it. If you want the system to be able to garbage collect images that were being tracked, make sure it can collect the MediaTracker instance as well.

Here’s an example that loads a seven-image slide show and displays a nice bar chart of the loading progress:
import java.util.*;
import java.applet.*;
import java.awt.**;

public class TrackedImageLoad extends Applet implements Runnable {
  MediaTracker tracker;
  int tracked;
  int frame_rate = 5;
  int current_img = 0;
  Thread motor;
  static final int MAXIMAGES = 10;
  Image img[] = new Image[MAXIMAGES];
  String name[] = new String[MAXIMAGES];
  volatile boolean stopFlag;

  public void init() {
    tracker = new MediaTracker(this);
    StringTokenizer st = new StringTokenizer(getParameter("img"),
    "+");

    while (st.hasMoreTokens() && tracked <= MAXIMAGES) {
      name[tracked] = st.nextToken();
      img[tracked] = getImage(getDocumentBase(),
      name[tracked] + ".jpg");
      tracker.addImage(img[tracked], tracked);
      tracked++;
    }
  }
}
public void paint(Graphics g) {
    String loaded = "";
    int donecount = 0;

    for(int i=0; i<tracked; i++) {
        if (tracker.checkID(i, true)) {
            donecount++;
            loaded += name[i] + " ";
        }
    }

    Dimension d = getSize();
    int w = d.width;
    int h = d.height;

    if (donecount == tracked) {
        frame_rate = 1;
        Image i = img[current_img++];
        int iw = i.getWidth(null);
        int ih = i.getHeight(null);
        g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
        if (current_img >= tracked)
            current_img = 0;
    } else {
        int x = w * donecount / tracked;
        g.setColor(Color.black);
        g.fillRect(0, h/3, x, 16);
        g.setColor(Color.white);
        g.fillRect(x, h/3, w-x, 16);
        g.setColor(Color.black);
        g.drawString(loaded, 10, h/2);
    }
}
This example creates a new MediaTracker in the init( ) method and then adds each of the named images as a tracked image with addImage( ). In the paint( ) method, it calls checkID( ) on each of the images that we’re tracking. If all of the images are loaded, they are displayed. If not, a simple bar chart of the number of images loaded is shown, with the names of the fully loaded images displayed underneath the bar. Figure 26-3 shows two scenes from this applet running. One is the bar chart, displaying that three of the images have been loaded. The other is the Van Gogh self-portrait during the slide show.
ImageProducer

ImageProducer is an interface for objects that want to produce data for images. An object that implements the ImageProducer interface will supply integer or byte arrays that represent image data and produce Image objects. As you saw earlier, one form of the createImage() method takes an ImageProducer object as its argument. There are two image producers contained in

Figure 26-3 Sample output from TrackedImageLoad
java.awt.image: MemoryImageSource and FilteredImageSource. Here, we will examine MemoryImageSource and create a new Image object from data generated in an applet.

MemoryImageSource

MemoryImageSource is a class that creates a new Image from an array of data. It defines several constructors. Here is the one we will be using:

```
MemoryImageSource(int width, int height, int pixel[ ],
                   int offset,
                   int scanLineWidth)
```

The MemoryImageSource object is constructed out of the array of integers specified by pixel, in the default RGB color model to produce data for an Image object. In the default color model, a pixel is an integer with Alpha, Red, Green, and Blue (0xAARRGGBB). The Alpha value represents a degree of transparency for the pixel. Fully transparent is 0 and fully opaque is 255. The width and height of the resulting image are passed in width and height. The starting point in the pixel array to begin reading data is passed in offset. The width of a scan line (which is often the same as the width of the image) is passed in scanLineWidth.

The following short example generates a MemoryImageSource object using a variation on a simple algorithm (a bitwise-exclusive-OR of the x and y address of each pixel) from the book Beyond Photography, The Digital Darkroom by Gerard J.
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
    Image img;
    public void init() {
        Dimension d = getSize();
        int w = d.width;
        int h = d.height;
        int pixels[] = new int[w * h];
        int i = 0;

        for(int y=0; y<h; y++) {
            for(int x=0; x<w; x++) {
                int r = (x^y)&0xff;
                int g = (x*2^y*2)&0xff;
                int b = (x*4^y*4)&0xff;
                pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
            }
        }
        img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}

The data for the new MemoryImageSource is created in the init( ) method. An array of integers is created to hold the pixel values; the data is generated in the nested for loops where the r, g, and b values get shifted into a pixel in the pixels array. Finally, createImage( ) is called with a new instance of a MemoryImageSource created from the raw pixel data as its parameter. Figure 26-4 shows the image when we run the applet. (It looks much
ImageConsumer

ImageConsumer is an interface for objects that want to take pixel data from images and supply it as another kind of data. This, obviously, is the opposite of ImageProducer, described earlier. An object that implements the ImageConsumer interface is going to create int or byte arrays that represent pixels from an Image object. We will examine the PixelGrabber class, which is a simple implementation of the ImageConsumer interface.

PixelGrabber

The PixelGrabber class is defined within java.lang.image. It is the inverse of the MemoryImageSource class. Rather than constructing an image from an array of pixel values, it takes an existing image and grabs the pixel array from it. To use PixelGrabber, you first create an array of ints big enough to hold the pixel data, and then you create a
PixelGrabber instance passing in the rectangle that you want to grab. Finally, you call `grabPixels()` on that instance.

The PixelGrabber constructor that is used in this chapter is shown here:

```java
PixelGrabber(Image imgObj, int left, int top, int width, int height, int pixel [], int offset, int scanLineWidth)
```

Here, `imgObj` is the object whose pixels are being grabbed. The values of `left` and `top` specify the upper-left corner of the rectangle, and `width` and `height` specify the dimensions of the rectangle from which the pixels will be obtained. The pixels will be stored in `pixel` beginning at `offset`. The width of a scan line (which is often the same as the width of the image) is passed in `scanLineWidth`.

`grabPixels()` is defined like this:

```java
boolean grabPixels()
    throws InterruptedException

boolean grabPixels(long milliseconds)
    throws InterruptedException
```

Both methods return `true` if successful and `false` otherwise. In the second form, `milliseconds` specifies how long the method will wait for the pixels. Both throw `InterruptedException` if execution is interrupted by another thread.
Here is an example that grabs the pixels from an image and then creates a histogram of pixel brightness. The histogram is simply a count of pixels that are a certain brightness for all brightness settings between 0 and 255. After the applet paints the image, it draws the histogram over the top.
public class HistoGrab extends Applet {
    Dimension d;
    Image img;
    int iw, ih;
    int pixels[];
    int w, h;
    int hist[] = new int[256];
    int max_hist = 0;

    public void init() {
        d = getSize();
        w = d.width;
        h = d.height;

        try {
            img = getImage(getDocumentBase(), getParameter("img"));
            MediaTracker t = new MediaTracker(this);

            t.addImage(img, 0);
            t.waitForID(0);
            iw = img.getWidth(null);
            ih = img.getHeight(null);
            pixels = new int[iw * ih];
            PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih, pixels, 0, iw);
            pg.grabPixels();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
            return;
        }
    }
}
for (int i=0; i<iw*ih; i++) {
    int p = pixels[i];
    int r = 0xff & (p >> 16);
    int g = 0xff & (p >> 8);
    int b = 0xff & (p);
    int y = (int) (.33 * r + .56 * g + .11 * b);
    hist[y]++;
}

for (int i=0; i<256; i++) {
    if (hist[i] > max_hist)
        max_hist = hist[i];
}

public void update() {}
Figure 26-5 Sample output from HistoGrab

ImageFilter

Given the ImageProducer and ImageConsumer interface pair—and their concrete classes MemoryImageSource and PixelGrabber—you can create an arbitrary set of translation filters that takes a source of pixels, modifies them, and passes them on to an arbitrary consumer. This mechanism is analogous to the way concrete classes are created from the abstract I/O classes InputStream, OutputStream, Reader, and Writer (described in Chapter 19). This stream model for images is completed by the introduction of the ImageFilter class. Some subclasses of ImageFilter in the java.awt.image package are AreaAveragingScaleFilter, CropImageFilter, ReplicateScaleFilter, and RGBImageFilter. There is also an implementation of ImageProducer called FilteredImageSource, which takes an arbitrary
ImageFilter and wraps it around an ImageProducer to filter the pixels it produces. An instance of FilteredImageSource can be used as an ImageProducer in calls to createImage(), in much the same way that BufferedInputStreams can be passed off as InputStreams.

In this chapter, we examine two filters: CropImageFilter and RGBImageFilter.

CropImageFilter

CropImageFilter filters an image source to extract a rectangular region. One situation in which this filter is valuable is where you want to use several small images from a single, larger source image. Loading twenty 2K images takes much longer than loading a single 40K image that has many frames of an animation tiled into it. If every subimage is the same size, then you can easily extract these images by using CropImageFilter to disassemble the block once your program starts. Here is an example that creates 16 images taken from a single image. The tiles are then scrambled by swapping a random pair from the 16 images 32 times.
/*
 * <applet code=TileImage.class width=288 height=399>
 * <param name=img value=picasso.jpg>
 * </applet>
 */

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
    Image img;
    Image cell[] = new Image[4*4];
    int iw, ih;
    int tw, th;

    public void init() {
        try {
            try {
                img = getImage(getDocumentBase(), getParameter("img"));
                MediaTracker t = new MediaTracker(this);
                t.addImage(img, 0);
                t.waitForID(0);
                iw = img.getWidth(null);
                ih = img.getHeight(null);
                tw = iw / 4;
                th = ih / 4;
                CropImageFilter f;
                FilteredImageSource fis;
                t = new MediaTracker(this);
                for (int y=0; y<4; y++) {
                    for (int x=0; x<4; x++) {
                        f = new CropImageFilter(tw*x, th*y, tw, th);
                        fis = new FilteredImageSource(img.getSource(), f);
                        int i = y*4+x;
                        cell[i] = createImage(fis);
                        t.addImage(cell[i], i);
                    }
                }
            }
        }
    }
}
Figure 26-6 shows a famous Picasso painting scrambled by the TileImage applet.

RGBImageFilter

The RGBImageFilter is used to convert one image to another, pixel by pixel, transforming the colors along the way. This filter could be used to brighten an image, to increase its contrast, or even to convert it to grayscale.

To demonstrate RGBImageFilter, we have developed a somewhat complicated example that employs a dynamic plug-in strategy for image-processing filters. We’ve created an interface for generalized image filtering so that an applet can simply load these filters based on <param> tags without having to know about all of the
ImageFilters in advance. This example consists of the main applet class called ImageFilterDemo, the interface called PlugInFilter, and a utility class called LoadedImage, which encapsulates some of the MediaTracker methods we’ve been using in this chapter. Also included are three filters—Grayscale, Invert, and Contrast—which simply manipulate the color space of the source image using RGBImageFilters, and two more classes—Blur and Sharpen—which do more complicated “convolution” filters that change pixel data based on the pixels surrounding each pixel of source data. Blur and Sharpen are subclasses of an abstract helper class called Convolver. Let’s look at each part of our example.

![Sample output from TileImage](image)

**Figure 26-6 Sample output from TileImage**

**ImageFilterDemo.java**
The ImageFilterDemo class is the applet framework for our sample image filters. It employs a simple
BorderLayout, with a Panel at the South position to hold the buttons that will represent each filter. A Label object occupies the North slot for informational messages about filter progress. The Center is where the image (which is encapsulated in the LoadedImage Canvas subclass, described later) is put. We parse the buttons/filters out of the filters <param> tag, separating them with +’s using a StringTokenizer.

The actionPerformed( ) method is interesting because it uses the label from a button as the name of a filter class that it tries to load with (PlugInFilter)
Class.forName(a).newInstance( ). This method is robust and takes appropriate action if the button does not correspond to a proper class that implements PlugInFilter.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
    Image img;
    PlugInFilter pif;
    Image fimg;
    Image curImg;
    LoadedImage lim;
    Label lab;
    Button reset;

    public void init() {
        setLayout(new BorderLayout());
        Panel p = new Panel();

        add(p, BorderLayout.SOUTH);
        reset = new Button("Reset");
        reset.addActionListener(this);
        p.add(reset);
        StringTokenizer st = new StringTokenizer(getParameter("filters"), "+");

        while(st.hasMoreTokens()) {
            Button b = new Button(st.nextToken());
            b.addActionListener(this);
            p.add(b);
        }
    }
}
Figure 26-7 shows what the applet looks like when it is first loaded using the applet tag shown at the top of this source file.

```java
lab = new Label("\n");
add(lab, BorderLayout.NORTH);

img = getImage(getDocumentBase(), getParameter("img"));
lim = new BufferedImage(img);
add(lim, BorderLayout.CENTER);

public void actionPerformed(ActionEvent ae) {
    String a = "\n"
    try {
        a = ae.getActionCommand();
        if (a.equals("Reset")) {
            lim.set(img);
            lab.setText("Normal");
        } else {
            pif = (PlugInFilter) Class.forName(a).newInstance();
            fimg = pif.filter(this, img);
            lim.set(fimg);
            lab.setText("Filtered: " + a);
        }
        repaint();
    } catch (ClassNotFoundException e) {
        lab.setText(a + " not found");
        lim.set(img);
        repaint();
    } catch (InstantiationException e) {
        lab.setText("couldn’t new " + a);
    } catch (IllegalAccessException e) {
        lab.setText("no access: " + a);
    }
}
```
PlugInFilter.java

PlugInFilter is a simple interface used to abstract image filtering. It has only one method, filter( ), which takes the applet and the source image and returns a new image that has been filtered in some way.

```java
interface PlugInFilter {
    java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

LoadedImage.java
LoadedImage is a convenient subclass of Canvas, which takes an image at construction time and synchronously loads it using MediaTracker. LoadedImage then behaves properly inside of LayoutManager control, because it overrides the getPreferredSize() and getMinimumSize() methods. Also, it has a method called set() that can be used to set a new Image to be displayed in this Canvas. That is how the filtered image is displayed after the plug-in is finished.
The Grayscale filter is a subclass of RGBImageFilter, which means that Grayscale can use itself as the ImageFilter parameter to FilteredImageSource’s constructor. Then all it needs to do is override filterRGB() to change the incoming color values. It takes the red,
green, and blue values and computes the brightness of the pixel, using the NTSC (National Television Standards Committee) color-to-brightness conversion factor. It then simply returns a gray pixel that is the same brightness as the color source.

```java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = (rgb >> 16) & 0xff;
        int g = (rgb >> 8) & 0xff;
        int b = rgb & 0xff;
        int k = (int) (.56 * g + .33 * r + .11 * b);
        return (0xff000000 | k << 16 | k << 8 | k);
    }
}
```

**Invert.java**

The Invert filter is also quite simple. It takes apart the red, green, and blue channels and then inverts them by subtracting them from 255. These inverted values are packed back into a pixel value and returned.

```java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = 0xff - (rgb >> 16) & 0xff;
        int g = 0xff - (rgb >> 8) & 0xff;
```
int b = 0xff - rgb & 0xff;
return (0xff000000 | r << 16 | g << 8 | b);
}

Figure 26-8 Using the Invert filter with ImageFilterDemo

Figure 26-8 shows the image after it has been run through the Invert filter.

Contrast.java

The Contrast filter is very similar to Grayscale, except its override of filterRGB( ) is slightly more complicated. The
algorithm it uses for contrast enhancement takes the red, green, and blue values separately and boosts them by 1.2 times if they are already brighter than 128. If they are below 128, then they are divided by 1.2. The boosted values are properly clamped at 255 by the multclamp() method.

```java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Contrast extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    private int multclamp(int in, double factor) {
        in = (int) (in * factor);
        return in > 255 ? 255 : in;
    }

    double gain = 1.2;
    private int cont(int in) {
        return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = cont((rgb >> 16) & 0xff);
        int g = cont((rgb >> 8) & 0xff);
        int b = cont(rgb & 0xff);
        return (0xff000000 | r < 16 | g < 8 | b);
    }
}
```

Figure 26-9 shows the image after Contrast is pressed.

Convolver.java
The abstract class Convolver handles the basics of a convolution filter by implementing the ImageConsumer interface to move the source pixels into an array called imgpixels. It also creates a second array called
newimgpixels for the filtered data. Convolution filters sample a small rectangle of pixels around each pixel in an image, called the convolution kernel. This area, 3 × 3 pixels in this demo, is used to decide how to change the center pixel in the area.

NOTE

The reason that the filter can’t modify the imgpixels array in place is that the next pixel on a scan line would try to use the original value for the previous pixel, which would have just been filtered away.
The two concrete subclasses, shown in the next section, simply implement the `convolve()` method, using `imgpixels` for source data and `newimgpixels` to store the result.

```java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PlugInFilter {
  int width, height;
  int imgpixels[], newimgpixels[];
  boolean imageReady = false;

  abstract void convolve(); // filter goes here...

  public Image filter(Applet a, Image in) {
    imageReady = false;
    in.getSource().startProduction(this);

    waitForImage();
    newimgpixels = new int[width*height];

    try {
      convolve();
    } catch (Exception e) {
      System.out.println("Convolver failed: "+ e);
      e.printStackTrace();
    }
    return a.createImage(
      new MemoryImageSource(width, height, newimgpixels, 0, width));
  }

  synchronized void waitForImage() {
    try {
      while(!imageReady) wait();
    } catch (Exception e) {
      System.out.println("Interrupted");
    }
  }
}
public void setProperties(java.util.Hashtable<?, ?> dummy) {}
public void setColorModel(ColorModel dummy) {}
public void setHints(int dummy) {}

public synchronized void imageComplete(int dummy) {
    imageReady = true;
    notifyAll();
}

public void setDimensions(int x, int y) {
    width = x;
    height = y;
    imgpixels = new int[x*y];
}

public void setPixels(int x1, int y1, int w, int h,
                      ColorModel model, byte pixels[], int off, int scansize) {
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
    for(y=y1; y<y2; y++) {
        sx = sy;
        for(x=x1; x<x2; x++) {
            pix = model.getRGB(pixels[sx++]);
            if((pix & 0xff000000) == 0)
                pix = 0x00ffffff;
            imgpixels[y*width+x] = pix;
        }
        sy += scansize;
    }
}
Blur.java

The Blur filter is a subclass of Convolver and simply runs through every pixel in the source image array, imgpixels, and computes the average of the $3 \times 3$ box surrounding it. The corresponding output pixel in newimgpixels is that average value.
public class Blur extends Convolver {
    public void convolve() {
        for(int y=1; y<height-1; y++) {
            for(int x=1; x<width-1; x++) {
                int rs = 0;
                int gs = 0;
                int bs = 0;
                for(int k=-1; k<=1; k++) {
                    for(int j=-1; j<=1; j++) {
                        int rgb = imgpixels[(y+k)*width+x+j];
                        int r = (rgb >> 16) & 0xff;
                        int g = (rgb >> 8) & 0xff;
                        int b = rgb & 0xff;
                        rs += r;
                        gs += g;
                        bs += b;
                    }
                }
                rs /= 9;
                gs /= 9;
                bs /= 9;
                newimgpixels[y*width+x] = (0xff000000 |
                                        rs << 16 | gs << 8 | bs);
            }
        }
    }
}

Figure 26-10 shows the applet after Blur.
Figure 26-10 Using the Blur filter with ImageFilterDemo

Sharpen.java

The Sharpen filter is also a subclass of Convolver and is (more or less) the inverse of Blur. It runs through every pixel in the source image array, imgpixels, and computes the average of the $3 \times 3$ box surrounding it, not counting the center. The corresponding output pixel in newimgpixels has the difference between the center pixel and the surrounding average added to it. This basically says that if a pixel is 30 brighter than its
surroundings, make it another 30 brighter. If, however, it is 10 darker, then make it another 10 darker. This tends to accentuate edges while leaving smooth areas unchanged.

public class Sharpen extends Convolver {

    private final int clamp(int c) {
        return (c > 255 ? 255 : (c < 0 ? 0 : c));
    }

    public void convolve() {
        int r0=0, g0=0, b0=0;

        for(int y=1; y<height-1; y++) {
            for(int x=1; x<width-1; x++) {
                int rs = 0;
                int gs = 0;
                int bs = 0;

                for(int k=-1; k<=1; k++) {
                    for(int j=-1; j<=1; j++) {
                        int rgb = imgpixels[(y+k)*width+x+j];
                        int r = (rgb >> 16) & 0xff;
                        int g = (rgb >> 8) & 0xff;
                        int b = rgb & 0xff;
                        if (j == 0 && k == 0) {
                            r = clamp(r * 1.4)
                            g = clamp(g * 1.4)
                            b = clamp(b * 1.4)
                        }
                    }
                }
            }
        }
    }
}
\[ r_0 = r; \]
\[ g_0 = g; \]
\[ b_0 = b; \]
\} else { 
\[ rs += r; \]
\[ gs += g; \]
\[ bs += b; \]
\}
\}

rs >>= 3;
gs >>= 3;
bs >>= 3;
newimgpixels[y*width+x] = (0xff000000 | 
  clamp(r0+r0-rs) << 16 | 
  clamp(g0+g0-gs) << 8 | 
  clamp(b0+b0-bs));
Figure 26-11 Using the Sharpen filter with ImageFilterDemo

Figure 26-11 shows the applet after Sharpen.

Cell Animation
Now that we have presented an overview of the image APIs, we can put together an interesting applet that will display a sequence of animation cells. The animation cells are taken from a single image that can arrange the cells in a grid specified via the rows and cols <param> tags. Each cell in the image is snipped out in a way
similar to that used in the TileImage example earlier. We obtain the sequence in which to display the cells from the sequence `<param>` tag. This is a comma-separated list of cell numbers that is zero-based and proceeds across the grid from left to right, top to bottom.

Once the applet has parsed the `<param>` tags and loaded the source image, it cuts the image into a number of small subimages. Then, a thread is started that causes the images to be displayed according to the order described in sequence. The thread sleeps for enough time to maintain the framerate. Here is the source code:

```java
// Animation example.
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;

public class Animation extends Applet implements Runnable {
    Image cell[];
    final int MAXSEQ = 64;
    int sequence[];
    int nseq;
    int idx;
    int framerate;
    volatile boolean stopFlag;

    private int intDef(String s, int def) {
        int n = def;
        if (s != null) {
            try {
                n = Integer.parseInt(s);
            } catch (NumberFormatException e) {
                System.out.println("Number Format Exception");
            }
        }
        return n;
    }

```
public void init() {
    framerate = intDef(getParameter("framerate"), 5);
    int tilex = intDef(getParameter("cols"), 1);
    int tiley = intDef(getParameter("rows"), 1);
    cell = new Image[tilex*tiley];

    StringTokenizer st = new StringTokenizer(getParameter("sequence"), ",");
    sequence = new int[MAXSEQ];
    nseq = 0;
    while (st.hasMoreTokens() && nseq < MAXSEQ) {
        sequence[nseq] = intDef(st.nextToken(), 0);
        nseq++;
    }

    try {
        Image img = getImage(getDocumentBase(), getParameter("img"));
        MediaTracker t = new MediaTracker(this);
        t.addImage(img, 0);
        t.waitForID(0);
        int iw = img.getWidth(null);
        int ih = img.getHeight(null);
        int tw = iw / tilex;
        int th = ih / tiley;
        CropImageFilter f;
        FilteredImageSource fis;
for (int y=0; y<tiley; y++) {
    for (int x=0; x<tilex; x++) {
        f = new CropImageFilter(tw*x, th*y, tw, th);
        fis = new FilteredImageSource(img.getSource(), f);
        int i = y*tilex+x;
        cell[i] = createImage(fis);
        t.addImage(cell[i], i);
    }
}
t.waitForAll();
} catch (InterruptedException e) {
    System.out.println("Image Load Interrupted");
}

public void update(Graphics g) { }

public void paint(Graphics g) {
    g.drawImage(cell[sequence[idx]], 0, 0, null);
}
The following applet tag shows the famous locomotion study by Eadweard Muybridge, which proved that horses do, indeed, get all four hooves off the ground at once. (Of course, you can substitute another image file in your own applet.)

```
<applet code=Animation width=67 height=48>
  <param name=img value=horse.gif>
  <param name=rows value=3>
  <param name=cols value=4>
  <param name=sequence value=0,1,2,3,4,5,6,7,8,9,10,11>
  <param name=framerate value=15>
</applet>
```

**Figure 26-12** shows the applet running. Notice the source image that has been loaded below the applet.
using a normal `<img>` tag.

Figure 26-12 Sample output of Animation

Additional Imaging Classes
In addition to the imaging classes described in this chapter, java.awt.image supplies several others that offer enhanced control over the imaging process and that support advanced imaging techniques. Also available is the imaging package called javax.imageio. This package supports plug-ins that handle various image formats. If sophisticated graphical output is of special interest to you, then you will want to explore the additional classes found in java.awt.image and javax.imageio.
From the start, Java has provided built-in support for multithreading and synchronization. For example, new threads can be created by implementing Runnable or by extending Thread; synchronization is available by use of the synchronized keyword; and interthread communication is supported by the wait( ) and notify( ) methods that are defined by Object. In general, this built-in support for multithreading was one of Java's most important innovations and is still one of its major strengths.

However, as conceptually pure as Java's original support for multithreading is, it is not ideal for all applications—especially those that make intensive use of multiple threads. For example, the original multithreading support does not provide several high-level features, such as semaphores, thread pools, and execution managers, that facilitate the creation of intensively concurrent programs.

It is important to explain at the outset that many Java programs make use of multithreading and are, therefore, "concurrent." For example, many applets and servlets use multithreading. However, as it is used in this chapter, the term concurrent program refers to a program that makes extensive, integral use of concurrently executing threads. An example of such a program is one that uses separate threads to simultaneously compute the partial results of a larger computation. Another example is a
program that coordinates the activities of several threads, each of which seeks access to information in a database. In this case, read-only accesses might be handled differently from those that require read/write capabilities.

To begin to handle the needs of a concurrent program, JDK 5 added the concurrency utilities, also commonly referred to as the concurrent API. The original set of concurrency utilities supplied many features that had long been wanted by programmers who develop concurrent applications. For example, it offered synchronizers (such as the semaphore), thread pools, execution managers, locks, several concurrent collections, and a streamlined way to use threads to obtain computational results.

Although the original concurrent API was impressive in its own right, it was significantly expanded by JDK 7. The most important addition is the Fork/Join Framework. The Fork/Join Framework facilitates the creation of programs that make use of multiple processors (such as those found in multicore systems). Thus, it streamlines the development of programs in which two or more pieces execute with true simultaneity (that is, true parallel execution), not just time-slicing. As you can easily imagine, parallel execution can dramatically increase the speed of certain operations. Because multicore systems are becoming commonplace, the inclusion of the Fork/Join Framework is as timely as it is powerful.

The original concurrent API was quite large, and the
Fork/Join Framework increases its size substantially. As you might expect, many of the issues surrounding the concurrency utilities are quite complex. It is beyond the scope of this book to discuss all of its facets. The preceding notwithstanding, it is important for all programmers to have a general, working knowledge of the concurrent API. Even in programs that are not intensively parallel, features such as synchronizers, callable threads, and executors, are applicable to a wide variety of situations. Perhaps most importantly, because of the rise of multicore computers, solutions involving the Fork/Join Framework will become more common. For these reasons, this chapter presents an overview of the concurrency utilities and shows several examples of their use. It concludes with an in-depth examination of the Fork/Join Framework.

The Concurrent API Packages

The concurrency utilities are contained in the java.util.concurrent package and in its two subpackages: java.util.concurrent.atomic and java.util.concurrent.locks. A brief overview of their contents is given here.

java.util.concurrent

java.util.concurrent defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication. It defines the following key features:

- Synchronizers
- Executors
Synchronizers offer high-level ways of synchronizing the interactions between multiple threads. The synchronizer classes defined by java.util.concurrent are

<table>
<thead>
<tr>
<th>Synchronizer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore</td>
<td>Implements the classic semaphore.</td>
</tr>
<tr>
<td>CountDownLatch</td>
<td>Waits until a specified number of events have occurred.</td>
</tr>
<tr>
<td>CyclicBarrier</td>
<td>Enables a group of threads to wait at a predefined execution point.</td>
</tr>
<tr>
<td>Exchanger</td>
<td>Exchanges data between two threads.</td>
</tr>
<tr>
<td>Phaser</td>
<td>Synchronizes threads that advance through multiple phases of an operation. (Added by JDK 7.)</td>
</tr>
</tbody>
</table>

Notice that each synchronizer provides a solution to a specific type of synchronization problem. This enables each synchronizer to be optimized for its intended use. In the past, these types of synchronization objects had to be crafted by hand. The concurrent API standardizes them and makes them available to all Java programmers.

Executors manage thread execution. At the top of the executor hierarchy is the Executor interface, which is used to initiate a thread. ExecutorService extends Executor and provides methods that manage execution. There are three implementations of ExecutorService: ThreadPoolExecutor, ScheduledThreadPoolExecutor, and ForkJoinPool (added by JDK 7). java.util.concurrent also defines the Executors utility class, which includes a number of static methods that simplify the creation of various executors.

Related to executors are the Future and Callable interfaces. A Future contains a value that is returned by a thread after it executes. Thus, its value becomes defined “in the future,” when the thread terminates. Callable
defines a thread that returns a value.

`java.util.concurrent` defines several concurrent collection classes, including `ConcurrentHashMap`, `ConcurrentLinkedQueue`, and `CopyOnWriteArrayList`. These offer concurrent alternatives to their related classes defined by the Collections Framework.

The Fork/Join Framework supports parallel programming. Its main classes are `ForkJoinTask`, `ForkJoinPool`, `RecursiveTask`, and `RecursiveAction`. As mentioned, the Fork/Join Framework was added by JDK 7.

Finally, to better handle thread timing, `java.util.concurrent` defines the `TimeUnit` enumeration.

`java.util.concurrent.atomic` facilitates the use of variables in a concurrent environment. It provides a means of efficiently updating the value of a variable without the use of locks. This is accomplished through the use of classes, such as `AtomicInteger` and `AtomicLong`, and methods, such as `compareAndSet()`, `decrementAndGet()`, and `getAndSet()`. These methods execute as a single, non-interruptible operation.

`java.util.concurrent.locks` provides an alternative to the use of synchronized methods. At the core of this alternative is the Lock interface, which defines the basic mechanism used to acquire and relinquish access to an object. The key methods are `lock()`, `tryLock()`, and
The advantage to using these methods is greater control over synchronization.

The remainder of this chapter takes a closer look at the constituents of the concurrent API.

Using Synchronization Objects
Synchronization objects are supported by the Semaphore, CountDownLatch, CyclicBarrier, Exchanger, and Phaser classes. Collectively, they enable you to handle several formerly difficult synchronization situations with ease. They are also applicable to a wide range of programs—even those that contain only limited concurrency. Because the synchronization objects will be of interest to nearly all Java programs, each is examined here in some detail.

Semaphore
The synchronization object that many readers will immediately recognize is Semaphore, which implements a classic semaphore. A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented. Otherwise, the thread will be
blocked until a permit can be acquired. When the thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented. If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java’s Semaphore class implements this mechanism.

Semaphore has the two constructors shown here:

```
Semaphore(int num)
Semaphore(int num, boolean how)
```

Here, num specifies the initial permit count. Thus, num specifies the number of threads that can access a shared resource at any one time. If num is one, then only one thread can access the resource at any one time. By default, waiting threads are granted a permit in an undefined order. By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

To acquire a permit, call the `acquire()` method, which has these two forms:

```
void acquire() throws InterruptedException
void acquire(int num) throws InterruptedException
```

The first form acquires one permit. The second form acquires num permits. Most often, the first form is used. If the permit cannot be granted at the time of the call, then the invoking thread suspends until the permit is available.

To release a permit, call `release()`, which has these
two forms:

```java
void release()
void release(int num)
```

The first form releases one permit. The second form releases the number of permits specified by num.

To use a semaphore to control access to a resource, each thread that wants to use that resource must first call `acquire()` before accessing the resource. When the thread is done with the resource, it must call `release()`. Here is an example that illustrates the use of a semaphore:

```java
// A simple semaphore example.
import java.util.concurrent.*;

class SemDemo {
    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);

        new IncThread(sem, "A");
        new DecThread(sem, "B");
    }
}

// A shared resource.
class Shared {
    static int count = 0;
}

// A thread of execution that increments count.
class IncThread implements Runnable {
    String name;
    Semaphore sem;

    IncThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }
```
public void run() {
    System.out.println("Starting " + name);
    try {
        // First, get a permit.
        System.out.println(name + " is waiting for a permit.");
        sem.acquire();
        System.out.println(name + " gets a permit.");

        // Now, access shared resource.
        for(int i=0; i < 5; i++) {
            Shared.count++;
            System.out.println(name + " : " + Shared.count);

            // Now, allow a context switch -- if possible.
            Thread.sleep(10);
        }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        // Release the permit.
        System.out.println(name + " releases the permit.");
        sem.release();
    }
}

// A thread of execution that decrements count.
class DecThread implements Runnable {
    String name;
    Semaphore sem;
    DecThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }
}
The output from the program is shown here. (The precise order in which the threads execute may vary.)

Starting A
A is waiting for a permit.
A gets a permit.
A: 1
Starting B
B is waiting for a permit.
A: 2
A: 3
A: 4
A: 5
A releases the permit.
B: 4
B: 3
B: 2
B: 1
B: 0
B releases the permit.
The program uses a semaphore to control access to the count variable, which is a static variable within the Shared class. Shared.count is incremented five times by the run( ) method of IncThread and decremented five times by DecThread. To prevent these two threads from accessing Shared.count at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access Shared.count, as the output shows.

In both IncThread and DecThread, notice the call to sleep( ) within run( ). It is used to “prove” that accesses to Shared.count are synchronized by the semaphore. In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, Shared.count is first incremented five times by IncThread and then decremented five times by DecThread. The increments and decrements are not intermixed.

Without the use of the semaphore, accesses to Shared.count by both threads would have occurred simultaneously, and the increments and decrements would be intermixed. To confirm this, try commenting out the calls to acquire( ) and release( ). When you run the program, you will see that access to Shared.count is no longer synchronized, and each thread accesses it as
soon as it gets a timeslice.

Although many uses of a semaphore are as straightforward as that shown in the preceding program, more intriguing uses are also possible. Here is an example. The following program reworks the producer/consumer program shown in Chapter 11 so that it uses two semaphores to regulate the producer and consumer threads, ensuring that each call to `put()` is followed by a corresponding call to `get()`:
// An implementation of a producer and consumer
// that use semaphores to control synchronization.

import java.util.concurrent.Semaphore;

class Q {
    int n;

    // Start with consumer semaphore unavailable.
    static Semaphore semCon = new Semaphore(0);
    static Semaphore semProd = new Semaphore(1);

    void get() {
        try {
            semCon.acquire();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

        System.out.println("Got: " + n);
        semProd.release();
    }

    void put(int n) {
        try {
            semProd.acquire();
        } catch(InterruptedException e) {
            System.out.println("InterruptedException caught");
        }

        this.n = n;
        System.out.println("Put: " + n);
        semCon.release();
    }
}
class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        for(int i=0; i < 20; i++) q.put(i);
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        for(int i=0; i < 20; i++) q.get();
    }
}

class ProdCon {
    public static void main(String args[]) {
        Q q = new Q();
        new Consumer(q);
        new Producer(q);
    }
}

A portion of the output is shown here:

Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
As you can see, the calls to put() and get() are synchronized. That is, each call to put() is followed by a call to get() and no values are missed. Without the semaphores, multiple calls to put() would have occurred without matching calls to get(), resulting in values being missed. (To prove this, remove the semaphore code and observe the results.)

The sequencing of put() and get() calls is handled by two semaphores: semProd and semCon. Before put() can produce a value, it must acquire a permit from semProd. After it has set the value, it releases semCon. Before get() can consume a value, it must acquire a permit from semCon. After it consumes the value, it releases semProd. This “give and take” mechanism ensures that each call to put() must be followed by a call to get().

Notice that semCon is initialized with no available permits. This ensures that put() executes first. The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

CountDownLatch

Sometimes you will want a thread to wait until one or more events have occurred. To handle such a situation, the concurrent API supplies CountDownLatch. A CountDownLatch is initially created with a count of the number of events that must occur before the latch is released. Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.
CountDownLatch has the following constructor:

CountDownLatch(int num)

Here, num specifies the number of events that must occur in order for the latch to open.

To wait on the latch, a thread calls await(), which has the forms shown here:

void await() throws InterruptedException
boolean await(long wait, TimeUnit tu) throws InterruptedException

The first form waits until the count associated with the invoking CountDownLatch reaches zero. The second form waits only for the period of time specified by wait. The units represented by wait are specified by tu, which is an object the TimeUnit enumeration. (TimeUnit is described later in this chapter.) It returns false if the time limit is reached, and true if the countdown reaches zero.

To signal an event, call the countDown() method, shown next:

void countDown()

Each call to countDown() decrements the count associated with the invoking object.

The following program demonstrates CountDownLatch. It creates a latch that requires five events to occur before it opens.
import java.util.concurrent.CountDownLatch;

class CDLDemo {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);

        System.out.println("Starting");

        new MyThread(cdl);

        try {
            cdl.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        System.out.println("Done");
    }
}

class MyThread implements Runnable {
    CountDownLatch latch;

    MyThread(CountDownLatch c) {
        latch = c;
        new Thread(this).start();
    }

    public void run() {
        for(int i = 0; i<5; i++) {
            System.out.println(i);
            latch.countDown(); // decrement count
        }
    }
}

The output produced by the program is shown here:

Starting
0
1
2

Inside main( ), a CountDownLatch called cdl is created with an initial count of five. Next, an instance of MyThread is created, which begins execution of a new thread. Notice that cdl is passed as a parameter to MyThread’s constructor and stored in the latch instance variable. Then, the main thread calls await( ) on cdl, which causes execution of the main thread to pause until cdl’s count has been decremented five times.

Inside the run( ) method of MyThread, a loop is created that iterates five times. With each iteration, the countDown( ) method is called on latch, which refers to cdl in main( ). After the fifth iteration, the latch opens, which allows the main thread to resume.

CountDownLatch is a powerful yet easy-to-use synchronization object that is appropriate whenever a thread must wait for one or more events to occur.

CyclicBarrier
A situation not uncommon in concurrent programming occurs when a set of two or more threads must wait at a predetermined execution point until all threads in the set have reached that point. To handle such a situation, the concurrent API supplies the CyclicBarrier class. It enables you to define a synchronization object that suspends until the specified number of threads has reached the barrier point.

CyclicBarrier has the following two constructors:
CyclicBarrier(int numThreads)
CyclicBarrier(int numThreads, Runnable action)

Here, numThreads specifies the number of threads that must reach the barrier before execution continues. In the second form, action specifies a thread that will be executed when the barrier is reached.

Here is the general procedure that you will follow to use CyclicBarrier. First, create a CyclicBarrier object, specifying the number of threads that you will be waiting for. Next, when each thread reaches the barrier, have it call await() on that object. This will pause execution of the thread until all of the other threads also call await(). Once the specified number of threads has reached the barrier, await() will return and execution will resume. Also, if you have specified an action, then that thread is executed.

The await() method has the following two forms:

    int await( ) throws InterruptedException, BrokenBarrierException

    int await(long wait, TimeUnit tu)

        throws InterruptedException, BrokenBarrierException, TimeoutException

The first form waits until all the threads have reached the barrier point. The second form waits only for the period of time specified by wait. The units represented by wait are specified by tu. Both forms return a value that indicates the order that the threads arrive at the barrier point. The first thread returns a value equal to
the number of threads waited upon minus one. The last thread returns zero.

Here is an example that illustrates CyclicBarrier. It waits until a set of three threads has reached the barrier. When that occurs, the thread specified by BarAction executes.

```java
// An example of CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
    public static void main(String args[]) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );
        System.out.println("Starting");

        new MyThread(cb, "A");
        new MyThread(cb, "B");
        new MyThread(cb, "C");
    }
}

// A thread of execution that uses a CyclicBarrier.

class MyThread implements Runnable {
    CyclicBarrier cbar;
    String name;

    MyThread(CyclicBarrier c, String n) {
        cbar = c;
        name = n;
        new Thread(this).start();
    }
}
public void run() {
    System.out.println(name);
    try {
        cbar.await();
    } catch (BrokenBarrierException exc) {
        System.out.println(exc);
    } catch (InterruptedException exc) {
        System.out.println(exc);
    }
}

// An object of this class is called when the CyclicBarrier ends.
class BarAction implements Runnable {
    public void run() {
        System.out.println("Barrier Reached!");
    }
}

The output is shown here. (The precise order in which the threads execute may vary.)

Starting
A
B
C
Barrier Reached!

A CyclicBarrier can be reused because it will release waiting threads each time the specified number of threads calls await(). For example, if you change main() in the preceding program so that it looks like this:

    public static void main(String args[]) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );
        System.out.println("Starting");
        new MyThread(cb, "A");
        new MyThread(cb, "B");
    }
The following output will be produced. (The precise order in which the threads execute may vary.)

Starting
A
B
C
Barrier Reached!
X
Y
Z
Barrier Reached!

As the preceding example shows, the CyclicBarrier offers a streamlined solution to what was previously a complicated problem.

Exchanger

Perhaps the most interesting of the synchronization classes is Exchanger. It is designed to simplify the exchange of data between two threads. The operation of an Exchanger is astoundingly simple: it simply waits until two separate threads call its exchange() method. When that occurs, it exchanges the data supplied by the threads. This mechanism is both elegant and easy to use. Uses for Exchanger are easy to imagine. For example, one thread might prepare a buffer for receiving information over a network connection. Another thread might fill that buffer with the information from the connection. The two threads work together so that each time a new buffer is needed, an exchange is made.
Exchanger is a generic class that is declared as shown here:

`Exchanger<V>`

Here, `V` specifies the type of the data being exchanged.

The only method defined by Exchanger is `exchange()`, which has the two forms shown here:

```java
V exchange(V buffer) throws InterruptedException
V exchange(V buffer, long wait, TimeUnit tu)
    throws InterruptedException, TimeoutException
```

Here, `buffer` is a reference to the data to exchange. The data received from the other thread is returned. The second form of `exchange()` allows a time-out period to be specified. The key point about `exchange()` is that it won’t succeed until it has been called on the same `Exchanger` object by two separate threads. Thus, `exchange()` synchronizes the exchange of the data.

Here is an example that demonstrates `Exchanger`. It creates two threads. One thread creates an empty buffer that will receive the data put into it by the second thread. Thus, the first thread exchanges an empty thread for a full one.
import java.util.concurrent.Exchanger;

class ExgrDemo {
    public static void main(String args[]) {
        Exchanger<String> exgr = new Exchanger<String>();

        new UseString(exgr);
        new MakeString(exgr);
    }
}

// A Thread that constructs a string.
class MakeString implements Runnable {
    Exchanger<String> ex;
    String str;

    MakeString(Exchanger<String> c) {
        ex = c;
        str = new String();

        new Thread(this).start();
    }

    public void run() {
        char ch = 'A';

        for(int i = 0; i < 3; i++) {
            // Fill Buffer
            for(int j = 0; j < 5; j++)
                str += ch++;

            try {
                // Exchange a full buffer for an empty one.
                str = ex.exchange(str);
            } catch(InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}
Here is the output produced by the program:

Got: ABCDE
Got: FGHIJ
Got: KLMNO

In the program, the main( ) method creates an Exchanger for strings. This object is then used to synchronize the exchange of strings between the MakeString and UseString classes. The MakeString class fills a string with data. The UseString exchanges an empty buffer for a full one. It then displays the contents of the newly constructed string. The exchange of empty and full buffers is synchronized by the exchange( ) method, which is called by both class’ run( ) method.

Phaser
JDK 7 adds a new synchronization class called Phaser. Its primary purpose is to enable the synchronization of threads that represent one or more phases of activity. For example, you might have a set of threads that implement three phases of an order-processing application. In the first phase, separate threads are used to validate customer information, check inventory, and confirm pricing. When that phase is complete, the second phase has two threads that compute shipping costs and all applicable tax. After that, a final phase confirms payment and determines estimated shipping time. In the past, to synchronize the multiple threads that comprise this scenario would require a bit of work on your part. With the inclusion of Phaser, the process is now much easier.

To begin, it helps to know that a Phaser works a bit like a CyclicBarrier, described earlier, except that it supports multiple phases. As a result, Phaser lets you define a synchronization object that waits until a specific phase has completed. It then advances to the next phase, again waiting until that phase concludes. It is important to understand that Phaser can also be used to synchronize only a single phase. In this regard, it acts much like a CyclicBarrier. However, its primary use is to synchronize multiple phases.

Phaser defines four constructors. Here are the two used in this section:

```java
Phaser()
Phaser(int numParties)
```
The first creates a phaser that has a registration count of zero. The second sets the registration count to numParties. The term party is often applied to the objects that register with a phaser. Although often there is a one-to-correspondence between the number of registrants and the number of threads being synchronized, this is not required. In both cases, the current phase is zero. That is, when a Phaser is created, it is initially at phase zero.

In general, here is how you use Phaser. First, create a new instance of Phaser. Next, register one or more parties with the phaser, either by calling register() or by specifying the number of parties in the constructor. For each registered party, have the phaser wait until all registered parties complete a phase. A party signals this by calling one of a variety of methods supplied by Phaser, such as arrive() or arriveAndAwaitAdvance(). After all parties have arrived, the phase is complete, and the phaser can move on to the next phase (if there is one), or terminate. The following sections explain the process in detail.

To register parties after a Phaser has been constructed, call register(). It is shown here:

```java
int register()
```

It returns the phase number of the phase to which it is registered.

To signal that a party has completed a phase, it must call arrive() or some variation of arrive(). When the number of arrivals equals the number of registered
parties, the phase is completed and the Phaser moves on the next phase (if there is one). The arrive() method has this general form:

    int arrive()

This method signals that a party (normally a thread of execution) has completed some task (or portion of a task). It returns the current phase number. If the phaser has been terminated, then it returns a negative value. The arrive() method does not suspend execution of the calling thread. This means that it does not wait for the phase to be completed. This method should be called only by a registered party.

If you want to indicate the completion of a phase and then wait until all other registrants have also completed that phase, use arriveAndAwaitAdvance(). It is shown here:

    int arriveAndAwaitAdvance()

It waits until all parties have arrived. It returns the next phase number or a negative value if the phaser has been terminated. This method should be called only by a registered party.

A thread can arrive and then deregister itself by calling arriveAndDeregister(). It is shown here:

    int arriveAndDeregister()

It returns the current phase number or a negative value if the phaser has been terminated. It does not wait until the phase is complete. This method should be called only by
a registered party.

To obtain the current phase number, call `getPhase()`, which is shown here:

```java
final int getPhase()
```

When a Phaser is created, the first phase will be 0, the second phase 1, the third phase 2, and so on. A negative value is returned if the invoking Phaser has been terminated.

Here is an example that shows Phaser in action. It creates three threads, each of which have three phases. It uses a Phaser to synchronize each phase.

```java
// An example of Phaser.

import java.util.concurrent.*;

class PhaserDemo {
    public static void main(String args[]) {
        Phaser phsr = new Phaser(1);
        int curPhase;

        System.out.println("Starting");

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // Wait for all threads to complete phase one.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");

        // Wait for all threads to complete phase two.
        curPhase = phsr.getPhase();
        phsr.arriveAndAwaitAdvance();
        System.out.println("Phase " + curPhase + " Complete");
    }
}
```
curPhase = phsr.getPhase();
phsr.arriveAndAwaitAdvance();
System.out.println("Phase " + curPhase + " Complete");

// Deregister the main thread.
phsr.arriveAndDeregister();

if(phsr.isTerminated())
    System.out.println("The Phaser is terminated");
}

// A thread of execution that uses a Phaser.
class MyThread implements Runnable {
    Phaser phsr;
    String name;

    MyThread(Phaser p, String n) {
        phsr = p;
        name = n;
        phsr.register();
        new Thread(this).start();
The output is shown here:

Starting
Thread A Beginning Phase One
Thread C Beginning Phase One
Thread B Beginning Phase One
Phase 0 Complete
Thread B Beginning Phase Two
Thread C Beginning Phase Two
Thread A Beginning Phase Two
Phase 1 Complete
Thread C Beginning Phase Three
Thread B Beginning Phase Three
Thread A Beginning Phase Three
Phase 2 Complete
The Phaser is terminated

Let's look closely at the key sections of the program. First, in main( ), a Phaser called phsr is created with an
initial party count of 1 (which corresponds to the main thread). Then three threads are started by creating three MyThread objects. Notice that MyThread is passed a reference to phsr (the phaser). The MyThread objects use this phaser to synchronize their activities. Next, main( ) calls getPhase( ) to obtain the current phase number (which is initially zero) and then calls arriveAndAwaitAdvance( ). This causes main( ) to suspend until phase zero has completed. This won’t happen until all MyThreads also call arriveAndAwaitAdvance( ). When this occurs, main( ) will resume execution, at which point it displays that phase zero has completed, and it moves on to phase two. This process repeats until all three phases have finished. Then, main( ) calls arriveAndDeregister( ). At that point, all three MyThreads have also deregistered. Since this results in there being no registered parties when the phaser advances to the next phase, the phaser is terminated.

Now look at MyThread. First, notice that the constructor is passed a reference to the phaser that it will use and then registers with the new thread as a party on that phaser. Thus, each new MyThread becomes a party registered with the passed-in phaser. Also notice that each thread has three phases. In this example, each phase consists of a placeholder that simply displays the name of the thread and what it is doing. Obviously, in real-world code, the thread would be performing more meaningful actions. Between the first two phases, the thread calls arriveAndAwaitAdvance( ). Thus, each
thread waits until all threads have completed the phase (and the main thread is ready). After all threads have arrived (including the main thread), the phaser moves on to the next phase. After the third phase, each thread deregisters itself with a call to `arriveAndDeregister()`.

As the comments in `MyThread` explain, the calls to `sleep()` are used for the purposes of illustration to ensure that the output is not jumbled because of the multithreading. They are not needed to make the phaser work properly. If you remove them, the output may look a bit jumbled, but the phases will still be synchronized correctly.

One other point: Although the preceding example used three threads that were all of the same type, this is not a requirement. Each party that uses a phaser can be unique, with each performing some separate task.

It is possible to take control of precisely what happens when a phase advance occurs. To do this, you must override the `onAdvance()` method. This method is called by the runtime when a Phaser advances from one phase to the next. It shown here:

```java
protected boolean onAdvance(int phase, int numParties)
```

Here, `phase` will contain the current phase number prior to being incremented and `numParties` will contain the number of registered parties. To terminate the phaser, `onAdvance()` must return `true`. To keep the phaser alive, `onAdvance()` must return `false`. The default version of `onAdvance()` returns `true` (thus terminating the phaser) when there are no registered parties. As a general rule,
One reason to override `onAdvance()` is to enable a phaser to execute a specific number of phases and then stop. The following example gives you the flavor of this usage. It creates a class called `MyPhaser` that extends `Phaser` so that it will run a specified number of phases. It does this by overriding the `onAdvance()` method. The `MyPhaser` constructor accepts one argument, which specifies the number of phases to execute. Notice that `MyPhaser` automatically registers one party. This behavior is useful in this example, but the needs of your own applications may differ.

```java
// Extend Phaser and override onAdvance() so that only a specific
// number of phases are executed.
import java.util.concurrent.*;

// Extend MyPhaser to allow only a specific number of phases
// to be executed.
class MyPhaser extends Phaser {
    int numPhases;

    MyPhaser(int parties, int phaseCount) {
        super(parties);
        numPhases = phaseCount - 1;
    }

    // Override onAdvance() to execute the specified
    // number of phases.
    protected boolean onAdvance(int p, int regParties) {
        // This println() statement is for illustration only.
        // Normally, onAdvance() will not display output.
        System.out.println("Phase " + p + " completed.\n");

        // If all phases have completed, return true
        if(p == numPhases || regParties == 0) return true;
    }
```
// Otherwise, return false.
return false;
}
}

class PhaserDemo2 {
    public static void main(String args[]) {

        MyPhaser phsr = new MyPhaser(1, 4);

        System.out.println("Starting\n");

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // Wait for the specified number of phases to complete.
        while(!phsr.isTerminated()) {
            phsr.arriveAndAwaitAdvance();
        }

        System.out.println("The Phaser is terminated");
    }
}
The output from the program is shown here:

Starting
Thread B Beginning Phase 0
Thread A Beginning Phase 0
Thread C Beginning Phase 0
Phase 0 completed.

Thread A Beginning Phase 1
Thread B Beginning Phase 1
Thread C Beginning Phase 1
Phase 1 completed.

Thread C Beginning Phase 2
Thread B Beginning Phase 2
Thread A Beginning Phase 2
Phase 2 completed.

Thread C Beginning Phase 3
Thread B Beginning Phase 3
Thread A Beginning Phase 3
Phase 3 completed.

The Phaser is terminated

Inside main( ), one instance of Phaser is created. It is passed 4 as an argument, which means that it will execute four phases and then stop. Next, three threads are created and then the following loop is entered:

```java
// Wait for the specified number of phases to complete.
while(!phsr.isTerminated()) {
    phsr.arriveAndAwaitAdvance();
}
```

This loop simply calls `arriveAndAwaitAdvance()` until the phaser is terminated. The phaser won’t terminate until the specified number of phases have been executed. In this case, the loop continues to execute until four phases have run. Next, notice that the threads also call `arriveAndAwaitAdvance()` within a loop that runs until the phaser is terminated. This means that they will execute until the specified number of phases has been completed.

Now, look closely at the code for `onAdvance()`. Each time `onAdvance()` is called, it is passed the current phase and the number of registered parties. If the current phase equals the specified phase, or if the number of registered parties is zero, `onAdvance()` returns true, thus stopping the phaser. This is accomplished with this line of code:

```java
// If all phases have completed, return true
if(p == numPhases || regParties == 0) return true;
```

As you can see, very little code is needed to
accommodate the desired outcome.

Before moving on, it useful to point out that you don’t necessarily need to explicitly extend Phaser as the previous example does to simply override onAdvance(). In some cases, more compact code can be created by using an anonymous inner class to override onAdvance().

Phaser has additional capabilities that may be of use in your applications. You can wait for a specific phase by calling awaitAdvance(), which is shown here:

```java
int awaitAdvance(int phase)
```

Here, phase indicates the phase number on which awaitAdvance() will wait until a transition to the next phase takes place. It will return immediately if the argument passed to phase is not equal to the current phase. It will also return immediately if the phaser is terminated. However, if phase is passed the current phase, then it will wait until the phase increments. This method should be called only by a registered party. There is also an interruptible version of this method called awaitAdvanceInterruptibly().

To register more than one party, call bulkRegister(). To obtain the number of registered parties, call getRegisteredParties(). You can also obtain the number of arrived parties and unarrived parties by calling getArrivedParties() and getUnarrivedParties(), respectively. To force the phaser to enter a terminated state, call forceTermination().

Phaser also lets you create a tree of phasers. This is
supported by two additional constructors, which let you specify the parent, and the `getParent()` method.

Using an Executor

The concurrent API supplies a feature called an executor that initiates and controls the execution of threads. As such, an executor offers an alternative to managing threads through the `Thread` class.

At the core of an executor is the `Executor` interface. It defines the following method:

```java
void execute(Runnable thread)
```

The thread specified by `thread` is executed. Thus, `execute()` starts the specified thread.

The `ExecutorService` interface extends `Executor` by adding methods that help manage and control the execution of threads. For example, `ExecutorService` defines `shutdown()`, shown here, which stops the invoking `ExecutorService`.

```java
void shutdown()
```

`ExecutorService` also defines methods that execute threads that return results, that execute a set of threads, and that determine the shutdown status. We will look at several of these methods a little later.

Also defined is the interface `ScheduledExecutorService`, which extends `ExecutorService` to support the scheduling of threads.

The concurrent API defines three predefined executor classes: `ThreadPoolExecutor` and
ScheduledThreadPoolExecutor, and ForkJoinPool. ThreadPoolExecutor implements the Executor and ExecutorService interfaces and provides support for a managed pool of threads. ScheduledThreadPoolExecutor also implements the ScheduledExecutorService interface to allow a pool of threads to be scheduled. ForkJoinPool implements the Executor and ExecutorService interfaces and is used by the Fork/Join Framework. It is described later in this chapter.

A thread pool provides a set of threads that is used to execute various tasks. Instead of each task using its own thread, the threads in the pool are used. This reduces the overhead associated with creating many separate threads. Although you can use ThreadPoolExecutor and ScheduledThreadPoolExecutor directly, most often you will want to obtain an executor by calling one of the following static factory methods defined by the Executors utility class. Here are some examples:

```java
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int numThreads)
static ScheduledExecutorService newScheduledThreadPool(int numThreads)
```

ewCachedThreadPool() creates a thread pool that adds threads as needed but reuses threads if possible.
newFixedThreadPool() creates a thread pool that consists of a specified number of threads.
newScheduledThreadPool() creates a thread pool that supports thread scheduling. Each returns a reference to
an ExecutorService that can be used to manage the pool.

A Simple Executor Example

Before going any further, a simple example that uses an executor will be of value. The following program creates a fixed thread pool that contains two threads. It then uses that pool to execute four tasks. Thus, four tasks share the two threads that are in the pool. After the tasks finish, the pool is shut down and the program ends.

```java
// A simple example that uses an Executor.
import java.util.concurrent.*;

class SimpExec {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);
        CountDownLatch cd12 = new CountDownLatch(5);
        CountDownLatch cd13 = new CountDownLatch(5);
        CountDownLatch cd14 = new CountDownLatch(5);
        ExecutorService es = Executors.newFixedThreadPool(2);

        System.out.println("Starting");

        // Start the threads.
        es.execute(new MyThread(cdl, "A"));
        es.execute(new MyThread(cd12, "B"));
        es.execute(new MyThread(cd13, "C"));
        es.execute(new MyThread(cd14, "D"));

        try {
            cdl.await();
            cd12.await();
            cd13.await();
            cd14.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
    }
}
```
The output from the program is shown here. (The precise order in which the threads execute may vary.)

Starting
A: 0
A: 1
A: 2
A: 3
A: 4
C: 0
C: 1
C: 2
C: 3
C: 4
D: 0
D: 1
D: 2
D: 3
D: 4
B: 0
As the output shows, even though the thread pool contains only two threads, all four tasks are still executed. However, only two can run at the same time. The others must wait until one of the pooled threads is available for use.

The call to `shutdown()` is important. If it were not present in the program, then the program would not terminate because the executor would remain active. To try this for yourself, simply comment out the call to `shutdown()` and observe the result.

Using Callable and Future

One of the most interesting features of the concurrent API is the `Callable` interface. This interface represents a thread that returns a value. An application can use `Callable` objects to compute results that are then returned to the invoking thread. This is a powerful mechanism because it facilitates the coding of many types of numerical computations in which partial results are computed simultaneously. It can also be used to run a thread that returns a status code that indicates the successful completion of the thread.

`Callable` is a generic interface that is defined like this:

```
interface Callable<V>
```

Here, `V` indicates the type of data returned by the task.
Callable defines only one method, `call()`, which is shown here:

```java
V call() throws Exception
```

Inside `call()`, you define the task that you want performed. After that task completes, you return the result. If the result cannot be computed, `call()` must throw an exception.

A Callable task is executed by an `ExecutorService`, by calling its `submit()` method. There are three forms of `submit()`, but only one is used to execute a Callable. It is shown here:

```java
<T> Future<T> submit(Callable<T> task)
```

Here, `task` is the Callable object that will be executed in its own thread. The result is returned through an object of type `Future`.

`Future` is a generic interface that represents the value that will be returned by a Callable object. Because this value is obtained at some future time, the name `Future` is appropriate. `Future` is defined like this:

```java
interface Future<V>
```

Here, `V` specifies the type of the result.

To obtain the returned value, you will call `Future`'s `get()` method, which has these two forms:

```java
V get()
```

throws `InterruptedException`, `ExecutionException`
V get(long wait, TimeUnit tu)
    throws InterruptedException, ExecutionException, TimeoutException

The first form waits for the result indefinitely. The second form allows you to specify a timeout period in wait. The units of wait are passed in tu, which is an object of the TimeUnit enumeration, described later in this chapter.

The following program illustrates Callable and Future by creating three tasks that perform three different computations. The first returns the summation of a value, the second computes the length of the hypotenuse of a right triangle given the length of its sides, and the third computes the factorial of a value. All three computations occur simultaneously.
import java.util.concurrent.ROOT;

class CallableDemo {
    public static void main(String args[]) {
        ExecutorService es = Executors.newFixedThreadPool(3);
        Future<Integer> f;
        Future<Double> f2;
        Future<Integer> f3;

        System.out.println("Starting");

        f = es.submit(new Sum(10));
        f2 = es.submit(new Hypot(3, 4));
        f3 = es.submit(new Factorial(5));

        try {
            System.out.println(f.get());
            System.out.println(f2.get());
            System.out.println(f3.get());
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
        catch (ExecutionException exc) {
            System.out.println(exc);
        }
    }
}
es.shutdown();
System.out.println("Done");
}

// Following are three computational threads.

class Sum implements Callable<Integer> {
    int stop;

    Sum(int v) { stop = v; }

    public Integer call() {
        int sum = 0;
        for(int i = 1; i <= stop; i++) {
            sum += i;
        }
        return sum;
    }
}

class Hypot implements Callable<Double> {
    double sidel1, side2;

    Hypot(double s1, double s2) {
        sidel1 = s1;
        side2 = s2;
    }
}
The concurrent API defines several methods that take an argument of type `TimeUnit`, which indicates a time-out period. `TimeUnit` is an enumeration that is used to specify the granularity (or resolution) of the timing. `TimeUnit` is defined within `java.util.concurrent`. It can be one of the following values:

- DAYS
- HOURS
- MINUTES
- SECONDS
- MICROSECONDS
Although TimeUnit lets you specify any of these values in calls to methods that take a timing argument, there is no guarantee that the system is capable of the specified resolution.

Here is an example that uses TimeUnit. The CallableDemo class, shown in the previous section, is modified as shown next to use the second form of get() that takes a TimeUnit argument.

```java
try {
    System.out.println(f.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f2.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f3.get(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
    System.out.println(exc);
} catch (ExecutionException exc) {
    System.out.println(exc);
} catch (TimeoutException exc) {
    System.out.println(exc);
}
```

In this version, no call to get() will wait more than 10 milliseconds.

The TimeUnit enumeration defines various methods that convert between units. These are shown here:

```java
long convert(long tval, TimeUnit tu)
long toMicros(long tval)
long toMillis(long tval)
long toNanos(long tval)
long toSeconds(long tval)
long toDays(long tval)
```
The `convert()` method converts `tval` into the specified unit and returns the result. The `to` methods perform the indicated conversion and return the result.

`TimeUnit` also defines the following timing methods:

```java
void sleep(long delay) throws InterruptedException
void timedJoin(Thread thrd, long delay) throws InterruptedException
void timedWait(Object obj, long delay) throws InterruptedException
```

Here, `sleep()` pauses execution for the specified delay period, which is specified in terms of the invoking enumeration constant. It translates into a call to `Thread.sleep()`. The `timedJoin()` method is a specialized version of `Thread.join()` in which `thrd` pauses for the time period specified by `delay`, which is described in terms of the invoking time unit. The `timedWait()` method is a specialized version of `Object.wait()` in which `obj` is waited on for the period of time specified by `delay`, which is described in terms of the invoking time unit.

The Concurrent Collections

As explained, the concurrent API defines several collection classes that have been engineered for concurrent operation. They include:

- `ArrayBlockingQueue`
ConcurrentHashMap
ConcurrentLinkedDeque (Added by JDK 7.)
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
DelayQueue
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue (Added by JDK 7.)
PriorityBlockingQueue
SynchronousQueue

These offer concurrent alternatives to their related classes defined by the Collections Framework. These collections work much like the other collections except that they provide concurrency support. Programmers familiar with the Collections Framework will have no trouble using these concurrent collections.

Locks
The java.util.concurrent.locks package provides support for locks, which are objects that offer an alternative to using synchronized to control access to a shared resource. In general, here is how a lock works. Before accessing a shared resource, the lock that protects that resource is acquired. When access to the resource is complete, the lock is released. If a second thread attempts to acquire the lock when it is in use by another thread, the second thread will suspend until the lock is released. In this
way, conflicting access to a shared resource is prevented.

Locks are particularly useful when multiple threads need to access the value of shared data. For example, an inventory application might have a thread that first confirms that an item is in stock and then decreases the number of items on hand as each sale occurs. If two or more of these threads are running, then without some form of synchronization, it would be possible for one thread to be in middle of a transaction when the second thread begins its transaction. The result could be that both threads would assume that adequate inventory exists, even if there is only sufficient inventory on hand to satisfy one sale. In this type of situation, a lock offers a convenient means of handling the needed synchronization.

All locks implement the Lock interface. The methods defined by Lock are shown in Table 27-1. In general, to acquire a lock, call lock(). If the lock is unavailable, lock() will wait. To release a lock, call unlock(). To see if a lock is available, and to acquire it if it is, call tryLock(). This method will not wait for the lock if it is unavailable. Instead, it returns true if the lock is acquired and false otherwise. The newCondition() method returns a Condition object associated with the lock. Using a Condition, you gain detailed control of the lock through methods such as await() and signal(), which provide functionality similar to Object.wait() and Object.notify().

java.util.concurrent.locks supplies an implementation of Lock called ReentrantLock. ReentrantLock implements
a reentrant lock, which is a lock that can be repeatedly entered by the thread that currently holds the lock. Of course, in the case of a thread reentering a lock, all calls to `lock()` must be offset by an equal number of calls to `unlock()`. Otherwise, a thread seeking to acquire the lock will suspend until the lock is not in use.

The following program demonstrates the use of a lock. It creates two threads that access a shared resource called `Shared.count`. Before a thread can access `Shared.count`, it must obtain a lock. After obtaining the lock, `Shared.count` is incremented and then, before releasing the lock, the thread sleeps. This causes the second thread to attempt to obtain the lock. However, because the lock is still held by the first thread, the second thread must wait until the first thread stops sleeping and releases the lock. The output shows that access to `Shared.count` is, indeed, synchronized by the lock.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void lock()</code></td>
<td>Waits until the invoking lock can be acquired.</td>
</tr>
<tr>
<td><code>void lockInterruptibly()</code></td>
<td>Waits until the invoking lock can be acquired, unless interrupted.</td>
</tr>
<tr>
<td>throws InterruptedException</td>
<td></td>
</tr>
<tr>
<td><code>Condition newCondition()</code></td>
<td>Returns a <code>Condition</code> object that is associated with the invoking lock.</td>
</tr>
<tr>
<td><code>boolean tryLock()</code></td>
<td>Attempts to acquire the lock. This method will not wait if the lock is unavailable. Instead, it returns <code>true</code> if the lock has been acquired and <code>false</code> if the lock is currently in use by another thread.</td>
</tr>
<tr>
<td><code>boolean tryLock(long wait, TimeUnit tu)</code></td>
<td>Attempts to acquire the lock. If the lock is unavailable, this method will wait no longer than the period specified by <code>wait</code>, which is in <code>tu</code> units. It returns <code>true</code> if the lock has been acquired and <code>false</code> if the lock cannot be acquired within the specified period.</td>
</tr>
<tr>
<td>throws InterruptedException</td>
<td></td>
</tr>
<tr>
<td><code>void unlock()</code></td>
<td>Releases the lock.</td>
</tr>
</tbody>
</table>

Table 27-1 The Lock Methods
// A simple lock example.

import java.util.concurrent.locks.*;

class LockDemo {

    public static void main(String args[]) {
        ReentrantLock lock = new ReentrantLock();

        new LockThread(lock, "A");
        new LockThread(lock, "B");
    }
}

// A shared resource.
class Shared {
    static int count = 0;
}

// A thread of execution that increments count.
class LockThread implements Runnable {
    String name;
    ReentrantLock lock;

    LockThread(ReentrantLock lk, String n) {
        lock = lk;
        name = n;
        new Thread(this).start();
    }
}
The output is shown here. (The precise order in which the threads execute may vary.)

Starting A
A is waiting to lock count.
A is locking count.
A: 1
A is sleeping.
Starting B
B is waiting to lock count.
A is unlocking count.
B is locking count.
B: 2
B is sleeping.
B is unlocking count.

java.util.concurrent.locks also defines the ReadWriteLock interface. This interface specifies a lock
that maintains separate locks for read and write access. This enables multiple locks to be granted for readers of a resource as long as the resource is not being written. ReentrantReadWriteLock provides an implementation of ReadWriteLock.

Atomic Operations

java.util.concurrent.atomic offers an alternative to the other synchronization features when reading or writing the value of some types of variables. This package offers methods that get, set, or compare the value of a variable in one uninterruptible (that is, atomic) operation. This means that no lock or other synchronization mechanism is required.

Atomic operations are accomplished through the use of classes, such as AtomicInteger and AtomicLong, and methods such as get(), set(), compareAndSet(), decrementAndGet(), and getAndSet(), which perform the action indicated by their names.

Here is an example that demonstrates how access to a shared integer can be synchronized by the use of AtomicInteger:
In the program, a static AtomicInteger named ai is created by Shared. Then, three threads of type AtomThread are created. Inside run( ), Shared.ai is modified by calling getAndSet( ). This method returns the previous value and then sets the value to the one passed as an argument. The use of AtomicInteger prevents two threads from writing to ai at the same time.
In general, the atomic operations offer a convenient (and possibly more efficient) alternative to the other synchronization mechanisms when only a single variable is involved.

Parallel Programming via the Fork/Join Framework

In recent years, an important new trend has emerged in software development: parallel programming. Parallel programming is the name commonly given to the techniques that take advantage of computers that contain two or more processors (multicore). As most readers will know, multicore computers are becoming commonplace. The advantage that multi-processor environments offer is the ability to significantly increase program performance. As a result, there has been a growing need for a mechanism that gives Java programmers a simple, yet effective way to make use of multiple processors in a clean, scalable manner. To answer this need, JDK 7 adds several new classes and interfaces that support parallel programming. They are commonly referred to as the Fork/Join Framework. It is one of most important additions that JDK 7 has made to the Java class library. The Fork/Join Framework is defined in the java.util.concurrent package.

The Fork/Join Framework enhances multithreaded programming in two important ways. First, it simplifies the creation and use of multiple threads. Second, it automatically makes use of multiple processors. In other words, by using the Fork/Join Framework you enable your applications to automatically scale to make use of the number of available processors. These two features
make the Fork/Join Framework the recommended approach to multithreading when parallel processing is desired.

Before continuing, it is important to point out the distinction between traditional multithreading and parallel programming. In the past, most computers had a single CPU and multithreading was primarily used to take advantage of idle time, such as when a program is waiting for user input. Using this approach, one thread can execute while another is waiting. In other words, on a single-CPU system, multithreading is used to allow two or more tasks to share the CPU. This type of multithreading is typically supported by an object of type Thread (as described in Chapter 11). Although this type of multithreading will always remain quite useful, it was not optimized for situations in which two or more CPUs are available (multicore computers).

When multiple CPUs are present, a second type of multithreading capability that supports true parallel execution is required. With two or more CPUs, it is possible to execute portions of a program simultaneously, with each part executing on its own CPU. This can be used to significantly speed up the execution of some types of operations, such as sorting, transforming, or searching a large array. In many cases, these types of operations can be broken down into smaller pieces (each acting on a portion of the array), and each piece can be run on its own CPU. As you can imagine, the gain in efficiency can be enormous. Simply put: Parallel programming will be part of nearly every
programmer’s future because it offers a way to dramatically improve program performance.

The Main Fork/Join Classes

The Fork/Join Framework is packaged in java.util.concurrent. At the core of the Fork/Join Framework are the following four classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForkJoinTask&lt;V&gt;</td>
<td>An abstract class that defines a task</td>
</tr>
<tr>
<td>ForkJoinPool</td>
<td>Manages the execution of ForkJoinTasks</td>
</tr>
<tr>
<td>RecursiveAction</td>
<td>A subclass of ForkJoinTask&lt;V&gt; for tasks that do not return values</td>
</tr>
<tr>
<td>RecursiveTask&lt;V&gt;</td>
<td>A subclass of ForkJoinTask&lt;V&gt; for tasks that return values</td>
</tr>
</tbody>
</table>

Here is how they relate. A ForkJoinPool manages the execution of ForkJoinTasks. ForkJoinTask is an abstract class that is extended by two other abstract classes: RecursiveAction and RecursiveTask. Typically, your code will extend these classes to create a task. Before looking at the process in detail, an overview of the key aspects of each class will be helpful.

ForkJoinTask <V>

ForkJoinTask <V> is an abstract class that defines a task that can be managed by a ForkJoinPool. The type parameter V specifies the result type of the task. ForkJoinTask differs from Thread in that ForkJoinTask represents lightweight abstraction of a task, rather than a thread of execution. ForkJoinTasks are executed by threads managed by a thread pool of type ForkJoinPool. This mechanism allows a large number of tasks to be managed by a small number of actual threads. Thus, ForkJoinTasks are very efficient when compared to threads.
ForkJoinTask defines many methods. At the core are fork( ) and join( ), shown here:

```java
final ForkJoinTask<V> fork()
final V join()
```

The fork( ) method submits the invoking task for asynchronous execution of the invoking task. This means that the thread that calls fork( ) continues to run. The fork( ) method returns this after the task is scheduled for execution. It can be executed only from within the computational portion of another ForkJoinTask, which is running within a ForkJoinPool. (You will see how to do this, shortly.) The join( ) method waits until the task on which it is called terminates. The result of the task is returned. Thus, through the use of fork( ) and join( ), you can start one or more new tasks and then wait for them to finish.

Another important ForkJoinTask method is invoke( ). It combines the fork and join operations into a single call because it begins a task and then waits for it to end. It is shown here:

```java
final V invoke()
```

The result of the invoking task is returned.

You can invoke more than one task at a time by using invokeAll( ). Two of its forms are shown here:

```java
static void invokeAll(ForkJoinTask<?> taskA,
                     ForkJoinTask<?> taskB)
```
static void invokeAll(ForkJoinTask<?> … taskList)

In the first case, taskA and taskB are executed. In the second case, all specified tasks are executed. In both cases, the calling thread waits until all of the specified tasks have terminated. The invokeAll() method can be executed only from within the computational portion of another ForkJoinTask, which is running within a ForkJoinPool.

RecursiveAction
A subclass of ForkJoinTask is RecursiveAction. This class encapsulates a task that does not return a result. Typically, your code will extend RecursiveAction to create a task that has a void return type. RecursiveAction specifies four methods, but only one is usually of interest: the abstract method called compute(). When you extend RecursiveAction to create a concrete class, you will put the code that defines the task inside compute(). The compute() method represents the computational portion of the task.

The compute() method is defined by RecursiveAction like this:

protected abstract void compute()

Notice that compute() is protected. This means that it can be called only by other methods of its class or subclass. Also, because it is abstract, it must be implemented by a subclass (unless that subclass is also abstract).

In general, RecursiveAction is used to implement a
Recursive, divide-and-conquer strategy for tasks that don’t return results. (See “The Divide-and-Conquer Strategy” later in this chapter.)

RecursiveTask<V>
Another subclass of ForkJoinTask is RecursiveTask<V>. This class encapsulates a task that returns a result. The result type is specified by V. Typically, your code will extend RecursiveTask<V> to create a task that returns a value. Like RecursiveAction, it too specifies four methods, but often only the abstract compute( ) method is used, which represents the computational portion of the task. When you extend RecursiveTask<V> to create a concrete class, put the code that represents the task inside compute( ). This code must also return the result of the task.

The compute( ) method is defined by RecursiveTask<V> like this:

protected abstract V compute( )

Notice that compute( ) is protected. This means that it can be called only by other methods of its class or subclass. Also, because it is abstract, it must be implemented by a subclass. When implemented, it must return the result of the task.

In general, RecursiveTask is used to implement a recursive, divide-and-conquer strategy for tasks that return results. (See “The Divide-and-Conquer Strategy” later in this chapter.)

ForkJoinPool
The execution of ForkJoinTasks takes place within a ForkJoinPool, which also manages the execution of the tasks. Therefore, in order to execute a ForkJoinTask, you must first have a ForkJoinPool.

ForkJoinPool defines several constructors. Here are two commonly used ones:

ForkJoinPool()

ForkJoinPool(int pLevel)

The first creates a default pool that supports a level of parallelism equal to the number of processors available in the system. The second lets you specify the level of parallelism. Its value must be greater than zero and not more than the limits of the implementation. The level of parallelism determines the number of threads that can execute concurrently. As a result, the level of parallelism effectively determines the number of tasks that can be executed simultaneously. (Of course, the number of tasks that can execute simultaneously cannot exceed the number of processors.) It is important to understand that the level of parallelism does not, however, limit the number of tasks that can be managed by the pool. A ForkJoinPool can manage many more tasks than its level of parallelism. Also, the level of parallelism is only a target. It is not a guarantee.

After you have created an instance of ForkJoinPool, you can start a task in a number of different ways. The first task started is often thought of as the main task. Frequently, the main task begins subtasks that are also
managed by the pool. One common way to begin a main
task is to call invoke( ) on the ForkJoinPool. It is shown
here:

\[
<T> \text{T invoke(ForkJoinTask}<T> \text{ task)}
\]

This method begins the task specified by task, and it
returns the result of the task. This means that the calling
code waits until invoke( ) returns.

To start a task without waiting for its completion, you
can use execute( ). Here is one of its forms:

\[
\text{void execute(ForkJoinTask}<?> \text{ task)}
\]

In this case, task is started, but the calling code does not
wait for its completion. Rather, the calling code
continues execution asynchronously.

ForkJoinPool manages the execution of its threads
using an approach called work-stealing. Each worker
thread maintains a queue of tasks. If one worker thread’s
queue is empty, it will take a task from another worker
thread. This adds to overall efficiency and helps maintain
a balanced load. (Because of demands on CPU time by
other processes in the system, even two worker threads
with identical tasks in their respective queues may not
complete at the same time.)

One other point: ForkJoinPool uses daemon threads.
A daemon thread is automatically terminated when all
user threads have terminated. Thus, there is no need to
explicitly shut down a ForkJoinPool. However, it is
possible to do so by calling shutdown( ).
The Divide-and-Conquer Strategy

As a general rule, uses of the Fork/Join Framework will employ a divide-and-conquer strategy that is based on recursion. This is why the two subclasses of ForkJoinTask are called RecursiveAction and RecursiveTask. It is anticipated that you will extend one of these classes when creating your own fork/join task.

The divide-and-conquer strategy is based on recursively dividing a task into smaller subtasks until the size of a subtask is small enough to be handled sequentially. For example, a task that applies a transform to each element in an array of N integers can be broken down into two subtasks in which each transforms half the elements in the array. That is, one subtask transforms the elements 0 to N/2, and the other transforms the elements N/2 to N. In turn, each subtask can be reduced to another set of subtasks, each transforming half of the remaining elements. This process of dividing the array will continue until a threshold is reached in which a sequential solution is faster than creating another division.

The advantage of the divide-and-conquer strategy is that the processing can occur in parallel. Therefore, instead of cycling through an entire array using a single thread, pieces of the array can be processed simultaneously. Of course, the divide-and-conquer approach works in many cases in which an array (or collection) is not present, but the most common uses involve some type of array, collection, or grouping of data.
One of the keys to best employing the divide-and-conquer strategy is correctly selecting the threshold at which sequential processing (rather than further division) is used. Typically, an optimal threshold is obtained through profiling the execution characteristics. However, very significant speed-ups will still occur even when a less-than-optimal threshold is used. It is, however, best to avoid overly large or overly small thresholds. At the time of this writing, the Java API documentation for ForkJoinTask<T> states that, as a rule-of-thumb, a task should perform somewhere between 100 and 10,000 computational steps.

It is also important to understand that the optimal threshold value is also affected by how much time the computation takes. If each computational step is fairly long, then smaller thresholds might be better. Conversely, if each computational step is quite short, then larger thresholds could yield better results. For applications that are to be run on a known system, with a known number of processors, you can use the number of processors to make informed decisions about the threshold value. However, for applications that will be running on a variety of systems, the capabilities of which are not known in advance, you can make no assumptions about the execution environment.

One other point: Although multiple processors may be available on a system, other tasks (and the operating system, itself) will be competing with your application for CPU time. Thus, it is important not to assume that your program will have unrestricted access to all CPUs.
Furthermore, different runs of the same program may display different run time characteristics because of varying task loads.

A Simple First Fork/Join Example

At this point, a simple example that demonstrates the Fork/Join Framework and the divide-and-conquer strategy will be helpful. Following is a program that transforms the elements in an array of double into their square roots. It does so via a subclass of RecursiveAction.

```java
// A simple example of the basic divide-and-conquer strategy.
// In this case, RecursiveAction is used.
import java.util.concurrent.*;
import java.util.*;

// A ForkJoinTask (via RecursiveAction) that transforms
// the elements in an array of doubles into their square roots.
class SqrtTransform extends RecursiveAction {
    // The threshold value is arbitrarily set at 1,000 in this example.
    // In real-world code, its optimal value can be determined by
    // profiling and experimentation.
    final int seqThreshold = 1000;

    // Array to be accessed.
    double[] data;

    // Determines what part of data to process.
    int start, end;

    SqrtTransform(double[] vals, int s, int e ) {
        data = vals;
        start = s;
        end = e;
    }
```
protected void compute() {

    // If number of elements is below the sequential threshold, 
    // then process sequentially.
    if((end - start) < seqThreshold) {
        // Transform each element into its square root.
        for(int i = start; i < end; i++) {
            data[i] = Math.sqrt(data[i]);
        }
    } 

    else {
        // Otherwise, continue to break the data into smaller pieces.

        // Find the midpoint.
        int middle = (start + end) / 2;

        // Invoke new tasks, using the subdivided data.
        invokeAll(new SqrtTransform(data, start, middle),
                  new SqrtTransform(data, middle, end));
    }
}
A portion of the original sequence:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

A portion of the transformed sequence (to four decimal places):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000

As you can see, the values of the array elements have been transformed into their square roots.

Let's look closely at how this program works. First, notice that SqrtTransform is a class that extends RecursiveAction. As explained, RecursiveAction extends ForkJoinTask for tasks that do not return results. Next, notice the final variable seqThreshold. This is the value
that determines when sequential processing will take place. This value is set (somewhat arbitrarily) to 1,000. Next, notice that a reference to the array to be processed is stored in `data` and that the fields `start` and `end` are used to indicate the boundaries of the elements to be accessed.

The main action of the program takes place in `compute()`. It begins by checking if the number of elements to be processed is below the sequential processing threshold. If it is, then those elements are processed (by computing their square root in this example). If the sequential processing threshold has not been reached, then two new tasks are started by calling `invokeAll()`. In this case, each subtask processes half the elements. As explained earlier, `invokeAll()` waits until both tasks return. After all of the recursive calls unwind, each element in the array will have been modified, with much of the action taking place in parallel (if multiple processors are available).

Understanding the Impact of the Level of Parallelism

Before moving on, it is important to understand the impact that the level of parallelism has on the performance of a fork/join task and how the parallelism and the threshold interact. The program shown in this section lets you experiment with different degrees of parallelism and threshold values. Assuming that you are using a multicore computer, then you can interactively observe the effect of these values.

In the preceding example, because the default `ForkJoinPool` constructor was used, the default level of
parallelism was used, which is equal to the number of processors in the system. However, you can specify the level of parallelism that you want. One way shown earlier is to specify it when you create a ForkJoinPool using this constructor:

\[
\text{ForkJoinPool}(\text{int} \ p\text{Level})
\]

Here, pLevel specifies the level of parallelism, which must be greater than zero and less than the implementation defined limit.

The following program creates a fork/join task that transforms an array of doubles. The transformation is arbitrary, but it is designed to consume several CPU cycles. This was done to ensure that the effects of changing the threshold or the level of parallelism would be more clearly displayed. To use the program, specify the threshold value and the level of parallelism on the command line. The program then runs the tasks. It also displays the amount of time it takes the tasks to run. To do this, it uses System.nanoTime(), which returns the value of the JVM’s high-resolution timer.
// A simple program that lets you experiment with the effects of
// changing the threshold and parallelism of a ForkJoinTask.
import java.util.concurrent.*;

// A ForkJoinTask (via RecursiveAction) that performs a
// a transform on the elements of an array of doubles.
class Transform extends RecursiveAction {

    // Sequential threshold, which is set by the constructor.
    int seqThreshold;

    // Array to be accessed.
    double[] data;

    // Determines what part of data to process.
    int start, end;

    Transform(double[] vals, int s, int e, int t) {
        data = vals;
        start = s;
        end = e;
        seqThreshold = t;
    }
}
protected void compute() {

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThreshold) {
        // The following code assigns an element at an even index the
        // square root of its original value. An element at an odd
        // index is assigned its cube root. This code is designed
        // to simply consume CPU time so that the effects of concurrent
        // execution are more readily observable.
        for(int i = start; i < end; i++) {
            if((data[i] % 2) == 0)
                data[i] = Math.sqrt(data[i]);
            else
                data[i] = Math.cbrt(data[i]);
        }
    }
    else {
        // Otherwise, continue to break the data into smaller pieces.

        // Find the midpoint.
        int middle = (start + end) / 2;

        // Invoke new tasks, using the subdivided data.
        invokeAll(new Transform(data, start, middle, seqThreshold),
                  new Transform(data, middle, end, seqThreshold));
    }
}
To use the program, specify the level of parallelism followed by the threshold limit. You should try experimenting with different values for each, observing the results. Remember, to be effective, you must run the code on a computer with at least two processors. Also,
understand that two different runs may (almost certainly will) produce different results because of the effect of other processes in the system consuming CPU time.

To give you an idea of the difference that parallelism makes, try this experiment. First, execute the program like this:

```java
java FJExperiment 1 1000
```

This requests 1 level of parallelism (essentially sequential execution) with a threshold of 1,000. Here is a sample run produced on a dual-core computer:

```
Level of parallelism: 1
Sequential threshold: 1000
Elapsed time: 259677487 ns
```

Now, specify 2 levels of parallelism like this:

```java
java FJExperiment 2 1000
```

Here is sample output from this run produced by same dual-core computer:

```
Level of parallelism: 2
Sequential threshold: 1000
Elapsed time: 169254472 ns
```

As is evident, adding parallelism substantially decreases execution time, thus increasing the speed of the program. You should experiment with varying the threshold and parallelism on your own computer. The results may surprise you.

There are two other methods that you might find useful when experimenting with the execution characteristics of a fork/join program. First, you can obtain the level of parallelism by calling `getParallelism()`...
int getParallelism( )

It returns the parallelism level currently in effect. Recall that, by default, this will equal the number of available processors. Second, you can obtain the number of processors available in the system by calling availableProcessors( ), which is defined by the Runtime class. It is shown here:

int availableProcessors( )

The value returned may change from one call to the next because of other system demands.

An Example that Uses RecursiveTask<V>
The two preceding examples are based on RecursiveAction, which means that they concurrently execute tasks that do not return results. To create a task that returns a result, use RecursiveTask. In general, solutions are designed in the same manner as just shown. The key difference is that the compute( ) method returns a result. Thus, you must aggregate the results, so that when the first invocation finishes, it returns the overall result. Another difference is that you will typically start a subtask by calling fork( ) and join( ) explicitly (rather than implicitly by calling invokeAll( ), for example).

The following program demonstrates RecursiveTask. It creates a task called Sum that returns the summation of the values in an array of double. In this example, the array consists of 5,000 elements. However, every other
value is negative. Thus, the first values in the array are 0, 
−1, 2, −3, 4, and so on.

// A simple example that uses RecursiveTask<V>.
import java.util.concurrent.*;

// A RecursiveTask that computes the summation of an array of doubles.
class Sum extends RecursiveTask<Double> {

    // The sequential threshold value.
    final int seqThresHold = 500;

    // Array to be accessed.
    double[] data;

    // Determines what part of data to process.
    int start, end;

    Sum(double[] vals, int s, int e) {
        data = vals;
        start = s;
        end = e;
    }

    // Find the summation of an array of doubles.
    protected Double compute() {
        double sum = 0;
// If number of elements is below the sequential threshold,
// then process sequentially.
if((end - start) < seqThresHold) {
  // Sum the elements.
  for(int i = start; i < end; i++) sum += data[i];
}
else {  // Otherwise, continue to break the data into smaller pieces.

  // Find the midpoint.
  int middle = (start + end) / 2;

  // Invoke new tasks, using the subdivided data.
  Sum subTaskA = new Sum(data, start, middle);
  Sum subTaskB = new Sum(data, middle, end);

  // Start each subtask by forking.
  subTaskA.fork();
  subTaskB.fork();

  // Wait for the subtasks to return, and aggregate the results.
  sum = subTaskA.join() + subTaskB.join();
}
// Return the final sum.
return sum;
}

// Demonstrate parallel execution.
class RecurTaskDemo {
  public static void main(String args[]) {
    // Create a task pool.
    ForkJoinPool fjp = new ForkJoinPool();

    double[] nums = new double[5000];

    // Initialize nums with values that alternate between
    // positive and negative.
    for(int i=0; i < nums.length; i++)
      nums[i] = (double) (((i%2) == 0) ? i : -i);

    Sum task = new Sum(nums, 0, nums.length);

    // Start the ForkJoinTasks. Notice that, in this case,
    // invoke() returns a result.
    double summation = fjp.invoke(task);

    System.out.println("Summation "+ summation);
  }
}
Here’s the output from the program:

```
Summation -2500.0
```

There are a couple of interesting items in this program. First, notice that the two subtasks are executed by calling `fork()` as shown here:

```java
subTaskA.fork();
subTaskB.fork();
```

In this case, `fork()` is used because it starts a task but does not wait for it to finish. (Thus, it asynchronously runs the task.) The result of each task is obtained by calling `join()`, as shown here:

```java
sum = subTaskA.join() + subTaskB.join();
```

This statement waits until each task ends. It then adds the results of each and assigns the total to `sum`. Thus, the summation of each subtask is added to the running total. Finally, `compute()` ends by returning `sum`, which will be the final total when the first invocation returns.

There are other ways to approach the handling of the asynchronous execution of the subtasks. For example, the following sequence uses `fork()` to start `subTaskA` and uses `invoke()` to start and wait for `subTaskB`:

```java
subTaskA.fork();
sum = subTaskA.join() + subTaskB.invoke();
```

Another alternative is to have `subTaskB` call `compute()` directly, as shown here:

```java
subTaskA.fork();
sum = subTaskA.join() + subTaskB.compute();
```
Executing a Task Asynchronously

The preceding programs have called `invoke()` on a ForkJoinPool to initiate a task. This approach is commonly used when the calling thread must wait until the task has completed (which is often the case) because `invoke()` does not return until the task has terminated. However, you can start a task asynchronously. In this approach, the calling thread continues to execute. Thus, both the calling thread and the task execute simultaneously. To start a task asynchronously, use `execute()`, which is also defined by ForkJoinPool. It has the two forms shown here:

```java
void execute(ForkJoinTask<?> task)
void execute(Runnable task)
```

In both forms, `task` specifies the task to run. Notice that the second form lets you specify a Runnable rather than a ForkJoinTask task. Thus, it forms a bridge between Java’s traditional approach to multithreading and the new Fork/Join Framework. It is important to remember that the threads used by a ForkJoinPool are daemon. Thus, they will end when the main thread ends. As a result, you may need to keep the main thread alive until the tasks have finished.

Cancelling a Task

A task can be cancelled by calling `cancel()`, which is defined by ForkJoinTask. It has this general form:

```java
boolean cancel(boolean interruptOK)
```
It returns true if the task on which it was called is cancelled. It returns false if the task was already cancelled, has already completed, or can’t be cancelled. At this time, the interruptOK parameter is not used by the default implementation. In general, cancel() is intended to be called from code outside the task because a task can easily cancel itself by returning.

You can determine if a task has been cancelled by calling isCancelled(), as shown here:

```java
final boolean isCancelled()
```

It returns true if the invoking task has been cancelled prior to completion and false otherwise.

Determining a Task’s Completion Status

In addition to isCancelled(), which was just described, ForkJoinTask includes two other methods that you can use to determine a task’s completion status. The first is isCompletedNormally(), which is shown here:

```java
final boolean isCompletedNormally()
```

It returns true if the invoking task completed normally, that is, if it did not throw an exception and it was not cancelled via a call to cancel(). It returns false otherwise.

The second is isCompletedAbnormally(), which is shown here:

```java
final boolean isCompletedAbnormally()
```

It returns true if the invoking task completed because it was cancelled or because it threw an exception. It returns
Restarting a Task

Normally, you cannot rerun a task. In other words, once a task completes, it cannot be restarted. However, you can reinitialize the state of the task (after it has completed) so it can be run again. This is done by calling `reinitialize()`, as shown here:

```java
void reinitialize()
```

This method resets the state of the invoking task. However, any modification made to any persistent data that is operated upon by the task will not be undone. For example, if the task modifies an array, then those modifications are not undone by calling `reinitialize()`.

Things to Explore

The preceding discussion presented the fundamentals of the Fork/Join Framework and described the most commonly used methods. However, Fork/Join is a rich framework that includes additional capabilities that give you extended control over concurrency. Although it is far beyond the scope of this book to examine all of the issues and nuances surrounding parallel programming and the Fork/Join Framework, a sampling of the other features provided by `ForkJoinTask` and `ForkJoinPool` are mentioned here.

A Sampling of Other ForkJoinTask Features

As mentioned, methods such as `invokeAll()` and `fork()` can be called only from within a `ForkJoinTask`. This is
usually an easy rule to abide by, but, in some cases, you may have code that can be executed from inside or outside a task. You can determine if your code is executing inside a task by calling inForkJoinPool().

You can convert a Runnable or Callable object into a ForkJoinTask by using the adapt() method defined by ForkJoinTask. It has three forms, one for converting a Callable, one for a Runnable that does not return a result, and one for a Runnable that does return a result. In the case of a Callable, the call() method is run. In the case of Runnable, the run() method is run.

You can obtain an approximate count of the number of tasks that are in the queue of the invoking thread by calling getQueuedTaskCount(). You can obtain an approximate count of how many tasks the invoking thread has in its queue that are in excess of the number of other threads in the pool that might “steal” them, by calling getSurplusQueuedTaskCount(). Remember, in the Fork/Join Framework, work-stealing is one way in which a high level of efficiency is obtained. Although this process is automatic, in some cases, the information may prove helpful in optimizing throughput.

ForkJoinTask defines two methods that begin with the prefix quietly. They are shown here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>final void quietlyJoin()</td>
<td>Joins a task, but does not return a result or throw an exception</td>
</tr>
<tr>
<td>final void quietlyInvoke()</td>
<td>Invokes a task, but does not return a result or throw an exception</td>
</tr>
</tbody>
</table>

In essence, these methods are similar to their non-quiet counterparts except they don’t return values or throw exceptions.
You can attempt to “un-invoke” (in other words, unschedule) a task by calling `tryUnfork()`.

`ForkJoinTask` implements `Serializable`. Thus, it can be serialized. However, serialization is not used during execution.

**A Sampling of Other ForkJoinPool Features**

One method that is quite useful when tuning fork/join applications is `ForkJoinPool`’s override of `toString()` method. It displays a “user-friendly” synopsis of the state of the pool. To see it in action, use this sequence to start and then wait for the task in the `FJExperiment` class of the task experimenter program shown earlier:

```java
// Asynchronously start the main ForkJoinTask.
fjp.execute(task);

// Display the state of the pool while waiting.
while(!task.isDone()) {
    System.out.println(fjp);
}
```

When you run the program, you will see a series of messages on the screen that describe the state of the pool. Here is an example of one. Of course, your output may vary, based on the number of processors, threshold values, task load, and so on.

```
java.util.concurrent.ForkJoinPool@141d683[Running, parallelism = 2, size = 2, active = 0, running = 2, steals = 0, tasks = 0, submissions = 1]
```

You can determine if a pool is currently idle by calling `isQuiescent()`. It returns `true` if the pool has no active threads and `false` otherwise.

You can obtain the number of worker threads currently in the pool by calling `getPoolSize()`. You can
obtain an approximate count of the active threads in the pool by calling `getActiveThreadCount()`.

To shut down a pool, call `shutdown()`. Currently active tasks will still be executed, but no new tasks can be started. To stop a pool immediately, call `shutdownNow()`. In this case, an attempt is made to cancel currently active tasks. You can determine if a pool is shut down by calling `isShutdown()`. It returns `true` if the pool has been shut down and `false` otherwise. To determine if the pool has been shut down and all tasks have been completed, call `isTerminated()`.

**Some Fork/Join Tips**

Here are a few tips to help you avoid some of the more troublesome pitfalls associated with using the Fork/Join Framework. First, avoid using a sequential threshold that is too low. In general, erring on the high side is better than erring on the low side. If the threshold is too low, more time can be consumed generating and switching tasks than in processing the tasks. Second, usually it is best to use the default level of parallelism. If you specify a smaller number, it may significantly reduce the benefits of using the Fork/Join Framework.

In general, a `ForkJoinTask` should not use synchronized methods or synchronized blocks of code. Also, you will not normally want to have the `compute()` method use other types of synchronization, such as a semaphore. (The new Phaser can, however, be used when appropriate because it is compatible with the fork/join mechanism.) Remember, the main idea behind
a ForkJoinTask is the divide-and-conquer strategy. Such an approach does not normally lend itself to situations in which outside synchronization is needed. Also, avoid situations in which substantial blocking will occur through I/O. Therefore, in general, a ForkJoinTask will not perform I/O. Simply put, to best utilize the Fork/Join Framework, a task should perform a computation that can run without outside blocking or synchronization.

One last point: Except under unusual circumstances, do not make assumptions about the execution environment that your code will run in. This means you should not assume that some specific number of processors will be available, or that the execution characteristics of your program won’t be affected by other processes running at the same time.

The Concurrency Utilities Versus Java’s Traditional Approach

Given the power and flexibility found in the new concurrency utilities, it is natural to ask the following question: Do they replace Java’s traditional approach to multithreading and synchronization? The answer is a resounding no! The original support for multithreading and the built-in synchronization features are still the mechanism that should be employed for many, many Java programs, applets, and servlets. For example, synchronized, wait( ), and notify( ) offer elegant solutions to a wide range of problems. However, when extra control is needed, the concurrency utilities are available to handle the chore. Furthermore, the new
Fork/Join Framework offers a powerful way to integrate parallel programming techniques into your more sophisticated applications.
When Java was originally released, it included a set of eight packages, called the core API. Each subsequent release added to the API. Today, the Java API contains a large number of packages. Many of the packages support areas of specialization that are beyond the scope of this book. However, four packages warrant an examination here: java.util.regex, java.lang.reflect, java.rmi, and java.text. They support regular expression processing, reflection, Remote Method Invocation (RMI), and text formatting, respectively.

The regular expression package lets you perform sophisticated pattern matching operations. This chapter provides an in-depth discussion of this package along with extensive examples. Reflection is the ability of software to analyze itself. It is an essential part of the Java Beans technology that is covered in Chapter 29.

Remote Method Invocation (RMI) allows you to build Java applications that are distributed among several machines. This chapter provides a simple client/server example that uses RMI. The text formatting capabilities of java.text have many uses. The one examined here formats date and time strings.

The Core Java API Packages

Table 28-1 lists all of the core API packages defined by
Java (those in the java namespace) and summarizes their functions.

<table>
<thead>
<tr>
<th>Package</th>
<th>Primary Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.applet</td>
<td>Supports construction of applets.</td>
</tr>
<tr>
<td>java.awt</td>
<td>Provides capabilities for graphical user interfaces.</td>
</tr>
<tr>
<td>java.awt.color</td>
<td>Supports color spaces and profiles.</td>
</tr>
<tr>
<td>java.awt.datatransfer</td>
<td>Transfers data to and from the system clipboard.</td>
</tr>
<tr>
<td>java.awt.dnd</td>
<td>Supports drag-and-drop operations.</td>
</tr>
<tr>
<td>java.awt.event</td>
<td>Handles events.</td>
</tr>
<tr>
<td>java.awt.font</td>
<td>Represents various types of fonts.</td>
</tr>
<tr>
<td>java.awt.geom</td>
<td>Allows you to work with geometric shapes.</td>
</tr>
<tr>
<td>java.awt.im</td>
<td>Allows input of Japanese, Chinese, and Korean characters to text editing components.</td>
</tr>
<tr>
<td>java.awt.im.spi</td>
<td>Supports alternative input devices.</td>
</tr>
<tr>
<td>java.awt.image</td>
<td>Processes images.</td>
</tr>
<tr>
<td>java.awt.image.renderable</td>
<td>Supports rendering-independent images.</td>
</tr>
<tr>
<td>java.awt.print</td>
<td>Supports general print capabilities.</td>
</tr>
<tr>
<td>java.beans</td>
<td>Allows you to build software components.</td>
</tr>
<tr>
<td>java.beans.beancontext</td>
<td>Provides an execution environment for Beans.</td>
</tr>
<tr>
<td>java.io</td>
<td>Inputs and outputs data.</td>
</tr>
<tr>
<td>java.lang</td>
<td>Provides core functionality.</td>
</tr>
<tr>
<td>java.lang.annotation</td>
<td>Supports annotations (metadata).</td>
</tr>
<tr>
<td>java.lang.instrument</td>
<td>Supports program instrumentation.</td>
</tr>
<tr>
<td>java.lang.invoke</td>
<td>Supports dynamic languages.</td>
</tr>
<tr>
<td>java.lang.management</td>
<td>Supports management of the execution environment.</td>
</tr>
<tr>
<td>java.lang.ref</td>
<td>Enables some interaction with the garbage collector.</td>
</tr>
<tr>
<td>java.lang.reflect</td>
<td>Analyzes code at run time.</td>
</tr>
<tr>
<td>java.math</td>
<td>Handles large integers and decimal numbers.</td>
</tr>
<tr>
<td>java.net</td>
<td>Supports networking.</td>
</tr>
<tr>
<td>java.nio</td>
<td>Top-level package for the NIO classes. Encapsulates buffers.</td>
</tr>
<tr>
<td>Package</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>java.nio.channels</td>
<td>Encapsulates channels, which are used by the NIO system.</td>
</tr>
<tr>
<td>java.nio.channels.spi</td>
<td>Supports service providers for channels.</td>
</tr>
<tr>
<td>java.nio.charset</td>
<td>Encapsulates character sets.</td>
</tr>
<tr>
<td>java.nio.charset.spi</td>
<td>Supports service providers for character sets.</td>
</tr>
<tr>
<td>java.nio.file</td>
<td>Provides NIO support for files.</td>
</tr>
<tr>
<td>java.nio.file.attribute</td>
<td>Supports NIO file attributes.</td>
</tr>
<tr>
<td>java.nio.file.spi</td>
<td>Supports NIO service providers for files.</td>
</tr>
<tr>
<td>java.rmi</td>
<td>Provides remote method invocation.</td>
</tr>
<tr>
<td>java.rmi.activation</td>
<td>Activates persistent objects.</td>
</tr>
<tr>
<td>java.rmi.dgc</td>
<td>Manages distributed garbage collection.</td>
</tr>
<tr>
<td>java.rmi.registry</td>
<td>Maps names to remote object references.</td>
</tr>
<tr>
<td>java.rmi.server</td>
<td>Supports remote method invocation.</td>
</tr>
<tr>
<td>java.security</td>
<td>Handles certificates, keys, digests, signatures, and other security functions.</td>
</tr>
<tr>
<td>java.security.acl</td>
<td>Manages access control lists.</td>
</tr>
<tr>
<td>java.security.cert</td>
<td>Parses and manages certificates.</td>
</tr>
<tr>
<td>java.security.interfaces</td>
<td>Defines interfaces for DSA (Digital Signature Algorithm) keys.</td>
</tr>
<tr>
<td>java.security.spec</td>
<td>Specifies keys and algorithm parameters.</td>
</tr>
<tr>
<td>java.sql</td>
<td>Communicates with a SQL (Structured Query Language) database.</td>
</tr>
<tr>
<td>java.text</td>
<td>Formats, searches, and manipulates text.</td>
</tr>
<tr>
<td>java.text.spi</td>
<td>Supports service providers for text formatting classes in java.text.</td>
</tr>
<tr>
<td>java.util</td>
<td>Contains common utilities.</td>
</tr>
<tr>
<td>java.util.concurrent</td>
<td>Supports the concurrent utilities.</td>
</tr>
<tr>
<td>java.util.concurrent.atomic</td>
<td>Supports atomic (that is, indivisible) operations on variables without the use of locks.</td>
</tr>
<tr>
<td>java.util.concurrent.locks</td>
<td>Supports synchronization locks.</td>
</tr>
<tr>
<td>java.util.jar</td>
<td>Creates and reads JAR files.</td>
</tr>
<tr>
<td>java.util.logging</td>
<td>Supports logging of information related to a program’s execution.</td>
</tr>
<tr>
<td>java.util.prefs</td>
<td>Encapsulates information relating to user preference.</td>
</tr>
<tr>
<td>java.util.regex</td>
<td>Supports regular expression processing.</td>
</tr>
<tr>
<td>java.util.spi</td>
<td>Supports service providers for the utility classes in java.util.</td>
</tr>
<tr>
<td>java.util.zip</td>
<td>Reads and writes compressed and uncompressed ZIP files.</td>
</tr>
</tbody>
</table>

Table 28-1 The Core Java API Packages

Regular Expression Processing
The java.util.regex package supports regular expression processing. As the term is used here, a regular expression
is a string of characters that describes a character sequence. This general description, called a pattern, can then be used to find matches in other character sequences. Regular expressions can specify wildcard characters, sets of characters, and various quantifiers. Thus, you can specify a regular expression that represents a general form that can match several different specific character sequences.

There are two classes that support regular expression processing: Pattern and Matcher. These classes work together. Use Pattern to define a regular expression. Match the pattern against another sequence using Matcher.

Pattern

The Pattern class defines no constructors. Instead, a pattern is created by calling the compile( ) factory method. One of its forms is shown here:

```
static Pattern compile(String pattern)
```

Here, pattern is the regular expression that you want to use. The compile( ) method transforms the string in pattern into a pattern that can be used for pattern matching by the Matcher class. It returns a Pattern object that contains the pattern.

Once you have created a Pattern object, you will use it to create a Matcher. This is done by calling the matcher( ) factory method defined by Pattern. It is shown here:
Matcher matcher(CharSequence str)

Here str is the character sequence that the pattern will be matched against. This is called the input sequence. CharSequence is an interface that defines a read-only set of characters. It is implemented by the String class, among others. Thus, you can pass a string to matcher().

Matcher

The Matcher class has no constructors. Instead, you create a Matcher by calling the matcher() factory method defined by Pattern, as just explained. Once you have created a Matcher, you will use its methods to perform various pattern matching operations.

The simplest pattern matching method is matches(), which simply determines whether the character sequence matches the pattern. It is shown here:

    boolean matches()

It returns true if the sequence and the pattern match, and false otherwise. Understand that the entire sequence must match the pattern, not just a subsequence of it.

To determine if a subsequence of the input sequence matches the pattern, use find(). One version is shown here:

    boolean find()

It returns true if there is a matching subsequence and
false otherwise. This method can be called repeatedly, allowing it to find all matching subsequences. Each call to find( ) begins where the previous one left off.

You can obtain a string containing the last matching sequence by calling group( ). One of its forms is shown here:

    String group( )

The matching string is returned. If no match exists, then an IllegalStateException is thrown.

You can obtain the index within the input sequence of the current match by calling start( ). The index one past the end of the current match is obtained by calling end( ). These methods are shown here:

    int start( )
    int end( )

Both throw IllegalStateException if no match exists.

You can replace all occurrences of a matching sequence with another sequence by calling replaceAll( ), shown here:

    String replaceAll(String newStr)

Here, newStr specifies the new character sequence that will replace the ones that match the pattern. The updated input sequence is returned as a string.

Regular Expression Syntax
Before demonstrating Pattern and Matcher, it is necessary to explain how to construct a regular expression. Although no rule is complicated by itself, there are a large number of them, and a complete discussion is beyond the scope of this chapter. However, a few of the more commonly used constructs are described here.

In general, a regular expression is comprised of normal characters, character classes (sets of characters), wildcard characters, and quantifiers. A normal character is matched as-is. Thus, if a pattern consists of “xy”, then the only input sequence that will match it is “xy”. Characters such as newline and tab are specified using the standard escape sequences, which begin with a \. For example, a newline is specified by \n. In the language of regular expressions, a normal character is also called a literal.

A character class is a set of characters. A character class is specified by putting the characters in the class between brackets. For example, the class [wxyz] matches w, x, y, or z. To specify an inverted set, precede the characters with a ^. For example, [^wxyz] matches any character except w, x, y, or z. You can specify a range of characters using a hyphen. For example, to specify a character class that will match the digits 1 through 9, use [1-9].

The wildcard character is the . (dot) and it matches any character. Thus, a pattern that consists of “.” will match these (and other) input sequences: “A”, “a”, “x”, and so on.
A quantifier determines how many times an expression is matched. The quantifiers are shown here:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Match one or more</td>
</tr>
<tr>
<td>*</td>
<td>Match zero or more</td>
</tr>
<tr>
<td>?</td>
<td>Match zero or one</td>
</tr>
</tbody>
</table>

For example, the pattern “x+” will match “x”, “xx”, and “xxx”, among others.

One other point: In general, if you specify an invalid expression, a PatternSyntaxException will be thrown.

Demonstrating Pattern Matching

The best way to understand how regular expression pattern matching operates is to work through some examples. The first, shown here, looks for a match with a literal pattern:

```
// A simple pattern matching demo.
import java.util.regex.*;

class RegExpr {
    public static void main(String args[]) {
        Pattern pat;
        Matcher mat;
        boolean found;

        pat = Pattern.compile("Java");
        mat = pat.matcher("Java");
        found = mat.matches(); // check for a match
        System.out.println("Testing Java against Java.");
        if(found) System.out.println("Matches");
        else System.out.println("No Match");

        System.out.println();

        System.out.println("Testing Java against Java 7.");
        mat = pat.matcher("Java 7"); // create a new matcher
```
found = mat.matches(); // check for a match

if(found) System.out.println("Matches");
else System.out.println("No Match");

The output from the program is shown here:

Testing Java against Java.
Matches

Testing Java against Java 7.
No Match

Let's look closely at this program. The program begins by creating the pattern that contains the sequence “Java”. Next, a Matcher is created for that pattern that has the input sequence “Java”. Then, the matches() method is called to determine if the input sequence matches the pattern. Because the sequence and the pattern are the same, matches() returns true. Next, a new Matcher is created with the input sequence “Java 7” and matches() is called again. In this case, the pattern and the input sequence differ, and no match is found. Remember, the matches() function returns true only when the input sequence precisely matches the pattern. It will not return true just because a subsequence matches.

You can use find() to determine if the input sequence contains a subsequence that matches the pattern. Consider the following program:

// Use find() to find a subsequence.
import java.util.regex.*;
class RegExpr2 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("Java");
        Matcher mat = pat.matcher("Java 7");

        System.out.println("Looking for Java in Java 7.");

        if(mat.find()) System.out.println("subsequence found");
        else System.out.println("No Match");
    }
}

The output is shown here:

Looking for Java in Java 7.
subsequence found

In this case, find( ) finds the subsequence "Java".

The find( ) method can be used to search the input sequence for repeated occurrences of the pattern because each call to find( ) picks up where the previous one left off. For example, the following program finds two occurrences of the pattern "test":

// Use find() to find multiple subsequences.
import java.util.regex.*;

class RegExpr3 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("test");
        Matcher mat = pat.matcher("test 1 2 3 test");

        while(mat.find()) {
            System.out.println("test found at index " +
                mat.start());
        }
    }
}

The output is shown here:

test found at index 0
As the output shows, two matches were found. The program uses the `start()` method to obtain the index of each match.

Using Wildcards and Quantifiers
Although the preceding programs show the general technique for using `Pattern` and `Matcher`, they don’t show their power. The real benefit of regular expression processing is not seen until wildcards and quantifiers are used. To begin, consider the following example that uses the `+` quantifier to match any arbitrarily long sequence of Ws:

```java
// Use a quantifier.
import java.util.regex.*;

class RegExpr4 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("W+");
        Matcher mat = pat.matcher("W WW WWW");

        while(mat.find())
            System.out.println("Match: " + mat.group());
    }
}
```

The output from the program is shown here:

```
Match: W
Match: WW
Match: WWW
```

As the output shows, the regular expression pattern "W+" matches any arbitrarily long sequence of Ws.

The next program uses a wildcard to create a pattern
that will match any sequence that begins with e and ends with d. To do this, it uses the dot wildcard character along with the + quantifier.

```
// Use wildcard and quantifier.
import java.util.regex.*;

class RegExpr5 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("e.+d");
        Matcher mat = pat.matcher("extend cup end table");

        while(mat.find())
            System.out.println("Match: " + mat.group());
    }
}
```

You might be surprised by the output produced by the program, which is shown here:

```
Match: extend cup end
```

Only one match is found, and it is the longest sequence that begins with e and ends with d. You might have expected two matches: “extend” and “end”. The reason that the longer sequence is found is that by default, `find()` matches the longest sequence that fits the pattern. This is called greedy behavior. You can specify reluctant behavior by adding the ? quantifier to the pattern, as shown in this version of the program. It causes the shortest matching pattern to be obtained.

```
// Use the ? quantifier.
import java.util.regex.*;

class RegExpr6 {
    public static void main(String args[]) {
        // Use reluctant matching behavior.
```
Pattern pat = Pattern.compile("e.+?d");
Matcher mat = pat.matcher("extend cup end table");

while(mat.find())
   System.out.println("Match: " + mat.group());
}

The output from the program is shown here:

Match: extend
Match: end

As the output shows, the pattern “e.+?d” will match the shortest sequence that begins with e and ends with d. Thus, two matches are found.

Working with Classes of Characters
Sometimes you will want to match any sequence that contains one or more characters, in any order, that are part of a set of characters. For example, to match whole words, you want to match any sequence of the letters of the alphabet. One of the easiest ways to do this is to use a character class, which defines a set of characters. Recall that a character class is created by putting the characters you want to match between brackets. For example, to match the lowercase characters a through z, use [a-z]. The following program demonstrates this technique:

// Use a character class.
import java.util.regex.*;

class RegExpr7 {
   public static void main(String args[]) {
      // Match lowercase words.
      Pattern pat = Pattern.compile("[a-z]+");  
      Matcher mat = pat.matcher("this is a test.");
   }
}
while (mat.find())
    System.out.println("Match: " + mat.group());
}

The output is shown here:

Match: this
Match: is
Match: a
Match: test

Using replaceAll( )
The replaceAll( ) method supplied by Matcher lets you perform powerful search and replace operations that use regular expressions. For example, the following program replaces all occurrences of sequences that begin with "Jon" with "Eric":

// Use replaceAll().
import java.util.regex.*;

class RegExpr8 {
    public static void main(String args[]) {
        String str = "Jon Jonathan Frank Ken Todd";

        Pattern pat = Pattern.compile("Jon.*? ");
        Matcher mat = pat.matcher(str);

        System.out.println("Original sequence: " + str);
        str = mat.replaceAll("Eric ");

        System.out.println("Modified sequence: " + str);
    }
}

The output is shown here:

Original sequence: Jon Jonathan Frank Ken Todd
Modified sequence: Eric Eric Frank Ken Todd
Because the regular expression “Jon.*? “ matches any string that begins with Jon followed by zero or more characters, ending in a space, it can be used to match and replace both Jon and Jonathan with the name Eric. Such a substitution is not possible without pattern matching capabilities.

Using split( )
You can reduce an input sequence into its individual tokens by using the split( ) method defined by Pattern. One form of the split( ) method is shown here:

    String[ ] split( CharSequence str )

It processes the input sequence passed in str, reducing it into tokens based on the delimiters specified by the pattern.

For example, the following program finds tokens that are separated by spaces, commas, periods, and exclamation points:

    // Use split().
    import java.util.regex.*;

    class RegExpr9 {
        public static void main(String args[]) {

            // Match lowercase words.
            Pattern pat = Pattern.compile( "[ ,.!]" );

            String strs[] = pat.split( "one two,alpha 12!done." );

            for(int i=0; i < strs.length; i++)
                System.out.println("Next token: " + strs[i]);
        }
    }

The output is shown here:

Next token: one
Next token: two
Next token: alpha9
Next token: 12
Next token: done

As the output shows, the input sequence is reduced to its individual tokens. Notice that the delimiters are not included.

Two Pattern-Matching Options
Although the pattern-matching techniques described in the foregoing offer the greatest flexibility and power, there are two alternatives which you might find useful in some circumstances. If you only need to perform a one-time pattern match, you can use the matches( ) method defined by Pattern. It is shown here:

    static boolean matches(String pattern, CharSequence str)

It returns true if pattern matches str and false otherwise. This method automatically compiles pattern and then looks for a match. If you will be using the same pattern repeatedly, then using matches( ) is less efficient than compiling the pattern and using the pattern-matching methods defined by Matcher, as described previously.

You can also perform a pattern match by using the matches( ) method implemented by String. It is shown
boolean matches(String pattern)

If the invoking string matches the regular expression in pattern, then matches() returns true. Otherwise, it returns false.

Exploring Regular Expressions

The overview of regular expressions presented in this section only hints at their power. Since text parsing, manipulation, and tokenization are a large part of programming, you will likely find Java’s regular expression subsystem a powerful tool that you can use to your advantage. It is, therefore, wise to explore the capabilities of regular expressions. Experiment with several different types of patterns and input sequences. Once you understand how regular expression pattern matching works, you will find it useful in many of your programming endeavors.

Reflection

Reflection is the ability of software to analyze itself. This is provided by the java.lang.reflect package and elements in Class. Reflection is an important capability, especially when using components called Java Beans. It allows you to analyze a software component and describe its capabilities dynamically, at run time rather than at compile time. For example, by using reflection, you can determine what methods, constructors, and fields a class
supports. Reflection was introduced in Chapter 12. It is examined further here.

The package java.lang.reflect includes several interfaces. Of special interest is Member, which defines methods that allow you to get information about a field, constructor, or method of a class. There are also eight classes in this package. These are listed in Table 28-2.

The following application illustrates a simple use of the Java reflection capabilities. It prints the constructors, fields, and methods of the class java.awt.Dimension. The program begins by using the forName( ) method of Class to get a class object for java.awt.Dimension.

<table>
<thead>
<tr>
<th>Class</th>
<th>Primary Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessibleObject</td>
<td>Allows you to bypass the default access control checks.</td>
</tr>
<tr>
<td>Array</td>
<td>Allows you to dynamically create and manipulate arrays.</td>
</tr>
<tr>
<td>Constructor</td>
<td>Provides information about a constructor.</td>
</tr>
<tr>
<td>Field</td>
<td>Provides information about a field.</td>
</tr>
<tr>
<td>Method</td>
<td>Provides information about a method.</td>
</tr>
<tr>
<td>Modifier</td>
<td>Provides information about class and member access modifiers.</td>
</tr>
<tr>
<td>Proxy</td>
<td>Supports dynamic proxy classes.</td>
</tr>
<tr>
<td>ReflectPermission</td>
<td>Allows reflection of private or protected members of a class.</td>
</tr>
</tbody>
</table>

Table 28-2 Classes Defined in java.lang.reflect

Once this is obtained, getConstructors( ), getFields( ), and getMethods( ) are used to analyze this class object. They return arrays of Constructor, Field, and Method objects that provide the information about the object. The Constructor, Field, and Method classes define several methods that can be used to obtain information about an object. You will want to explore these on your own.
However, each supports the toString() method. Therefore, using Constructor, Field, and Method objects as arguments to the println() method is straightforward, as shown in the program.

```java
// Demonstrate reflection.
import java.lang.reflect.*;
public class ReflectionDemo1 {
    public static void main(String args[]) {
        try {
            Class<?> c = Class.forName("java.awt.Dimension");
            System.out.println("Constructors:");
            Constructor constructors[] = c.getConstructors();
            for(int i = 0; i < constructors.length; i++) {
                System.out.println(" " + constructors[i]);
            }

            System.out.println("Fields:");
            Field fields[] = c.getFields();
            for(int i = 0; i < fields.length; i++) {
                System.out.println(" " + fields[i]);
            }

            System.out.println("Methods:");
            Method methods[] = c.getMethods();
            for(int i = 0; i < methods.length; i++) {
                System.out.println(" " + methods[i]);
            }
        } catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

Here is the output from this program. (The precise order may differ slightly from that shown.)

Constructors:
- public java.awt.Dimension(int,int)
- public java.awt.Dimension()
- public java.awt.Dimension(java.awt.Dimension)

Fields:
The next example uses Java’s reflection capabilities to obtain the public methods of a class. The program begins by instantiating class A. The `getClass()` method is applied to this object reference, and it returns the `Class` object for class A. The `getDeclaredMethods()` method returns an array of `Method` objects that describe only the methods declared by this class. Methods inherited from superclasses such as `Object` are not included.

Each element of the methods array is then processed. The `getModifiers()` method returns an int containing flags that describe which modifiers apply for this element. The `Modifier` class provides a set of is X methods, shown in Table 28-3, that can be used to
examine this value. For example, the static method 
isPublic( ) returns true if its argument includes the public modifier. Otherwise, it returns false. In the following program, if the method supports public access, its name is obtained by the getName( ) method and is then printed.

```java
// Show public methods.
import java.lang.reflect.*;
public class ReflectionDemo2 {
  public static void main(String args[]) {
    try {
      A a = new A();
      Class<?> c = a.getClass();
      System.out.println("Public Methods: ");
      Method methods[] = c.getDeclaredMethods();
      for(int i = 0; i < methods.length; i++) {
        int modifiers = methods[i].getModifiers();
        if(Modifier.isPublic(modifiers)) {
          System.out.println(" " + methods[i].getName());
        }
      }
    } catch(Exception e) {
      System.out.println("Exception: "+ e);
    }
  }
}

class A {
  public void a1() {
  }
  public void a2() {
  }
  protected void a3() {
  }
  private void a4() {
  }
}
Here is the output from this program:

```
Public Methods:
    a1
    a2
```

Beginning with JDK 7, Modifier also includes a set of static methods that return the type of modifiers that can be applied to a specific type of program element. These methods are

```
static int classModifiers( )
static int constructorModifiers( )
static int fieldModifiers( )
static int interfaceModifiers( )
static int methodModifiers( )
```

For example, methodModifiers( ) returns the modifiers that can be applied to a method. Each method returns flags, packed into an int, that indicate which modifiers are legal. The modifier values are defined by constants in Modifier, which include PROTECTED, PUBLIC, PRIVATE, STATIC, FINAL, and so on.
Remote Method Invocation (RMI)

Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications. While a complete discussion of RMI is outside the scope of this book, the following simplified example describes the basic principles involved.

---

Table 28-3 The “is” Methods Defined by Modifier That Determine Modifiers

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static boolean isAbstract(int val)</td>
<td>Returns true if val has the abstract flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isFinal(int val)</td>
<td>Returns true if val has the final flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isInterface(int val)</td>
<td>Returns true if val has the interface flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isNative(int val)</td>
<td>Returns true if val has the native flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isPrivate(int val)</td>
<td>Returns true if val has the private flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isProtected(int val)</td>
<td>Returns true if val has the protected flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isPublic(int val)</td>
<td>Returns true if val has the public flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isStatic(int val)</td>
<td>Returns true if val has the static flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isStrict(int val)</td>
<td>Returns true if val has the strict flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isSynchronized(int val)</td>
<td>Returns true if val has the synchronized flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isTransient(int val)</td>
<td>Returns true if val has the transient flag set and false otherwise.</td>
</tr>
<tr>
<td>static boolean isVolatile(int val)</td>
<td>Returns true if val has the volatile flag set and false otherwise.</td>
</tr>
</tbody>
</table>
This section provides step-by-step directions for building a simple client/server application by using RMI. The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

Step One: Enter and Compile the Source Code
This application uses four source files. The first file, AddServerIntf.java, defines the remote interface that is provided by the server. It contains one method that accepts two double arguments and returns their sum. All remote interfaces must extend the Remote interface, which is part of java.rmi.Remote defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a RemoteException.

```java
import java.rmi.*;

public interface AddServerIntf extends Remote {
    double add(double d1, double d2) throws RemoteException;
}
```

The second source file, AddServerImpl.java, implements the remote interface. The implementation of the add( ) method is straightforward. Remote objects typically extend UnicastRemoteObject, which provides functionality that is needed to make objects available from remote machines.
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject
    implements AddServerIntf {
    public AddServerImpl() throws RemoteException {
    }
    public double add(double d1, double d2) throws RemoteException {
        return d1 + d2;
    }
}

The third source file, AddServer.java, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the rebind() method of the Naming class (found in java.rmi). That method associates a name with an object reference. The first argument to the rebind() method is a string that names the server as “AddServer”. Its second argument is a reference to an instance of AddServerImpl.

import java.net.*;
import java.rmi.*;
public class AddServer {
    public static void main(String args[]) {
        try {
            AddServerImpl addServerImpl = new AddServerImpl();
            Naming.rebind("AddServer", addServerImpl);
        }
        catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}

The fourth source file, AddClient.java, implements the
client side of this distributed application. AddClient.java requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the rmi protocol. The string includes the IP address or name of the server and the string “AddServer”. The program then invokes the lookup( ) method of the Naming class. This method accepts one argument, the rmi URL, and returns a reference to an object of type AddServerIntf. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote add( ) method. The sum is returned from this method and is then printed.

```java
import java.rmi.*;

public class AddClient {
    public static void main(String args[]) {
        try {
            String addServerURL = "rmi://" + args[0] + "/AddServer";
            AddServerIntf addServerIntf =
                (AddServerIntf)Naming.lookup(addServerURL);
            System.out.println("The first number is: " + args[1]);
            double d1 = Double.valueOf(args[1]).doubleValue();
            System.out.println("The second number is: " + args[2]);
            double d2 = Double.valueOf(args[2]).doubleValue();
            System.out.println("The sum is: " + addServerIntf.add(d1, d2));
        }
        catch(Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}```
After you enter all the code, use javac to compile the four source files that you created.

Step Two: Manually Generate a Stub if Required

In the context of RMI, a stub is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

A remote method may accept arguments that are simple types or objects. In the latter case, the object may have references to other objects. All of this information must be sent to the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. Recall from Chapter 19 that the serialization facilities also recursively process all referenced objects.

If a response must be returned to the client, the process works in reverse. Note that the serialization and deserialization facilities are also used if objects are returned to a client.

Prior to Java 5, stubs needed to be built manually by using rmic. This step is not required for modern versions of Java. However, if you are working in a legacy environment, then you can use the rmic compiler, as shown here, to build a stub:
This command generates the file AddServerImplStub.class. When using rmic, be sure that CLASSPATH is set to include the current directory.

Step Three: Install Files on the Client and Server Machines
Copy AddClient.class, AddServerImplStub.class (if needed), and AddServerIntf.class to a directory on the client machine. Copy AddServerIntf.class, AddServerImpl.class, AddServerImplStub.class (if needed), and AddServer.class to a directory on the server machine.

NOTE
RMI has techniques for dynamic class loading, but they are not used by the example at hand. Instead, all of the files that are used by the client and server applications must be installed manually on those machines.

Step Four: Start the RMI Registry on the Server Machine
The JDK provides a program called rmiregistry, which executes on the server machine. It maps names to object references. First, check that the CLASSPATH environment variable includes the directory in which your files are located. Then, start the RMI Registry from the command line, as shown here:
start rmiregistry

When this command returns, you should see that a new window has been created. You need to leave this window open until you are done experimenting with the RMI example.

Step Five: Start the Server

The server code is started from the command line, as shown here:

   java AddServer

Recall that the AddServer code instantiates AddServerImpl and registers that object with the name “AddServer”.

Step Six: Start the Client

The AddClient software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

   java AddClient server1 8 9
   java AddClient 11.12.13.14 8 9

In the first line, the name of the server is provided. The second line uses its IP address (11.12.13.14).

You can try this example without actually having a remote server. To do so, simply install all of the
programs on the same machine, start rmiregistry, start AddServer, and then execute AddClient using this command line:

    java AddClient 127.0.0.1 8 9

Here, the address 127.0.0.1 is the “loop back” address for the local machine. Using this address allows you to exercise the entire RMI mechanism without actually having to install the server on a remote computer.

In either case, sample output from this program is shown here:

  The first number is: 8
  The second number is: 9
  The sum is: 17.0

NOTE

When working with RMI in the real world, it may be necessary for the server to install a security manager.

Text Formatting

The package java.text allows you to format, search, and manipulate text. Chapter 33 illustrates its NumberFormat class, which is used to format numeric data. This section examines two more of its most commonly used classes: those that format date and time information.

DateFormat Class

DateFormat is an abstract class that provides the ability to format and parse dates and times. The
The `getDateInstance( )` method returns an instance of `DateFormat` that can format date information. It is available in these forms:

```java
static final DateFormat getDateInstance( )
static final DateFormat getDateInstance(int style)
static final DateFormat getDateInstance(int style, Locale locale)
```

The argument `style` is one of the following values: `DEFAULT`, `SHORT`, `MEDIUM`, `LONG`, or `FULL`. These are `int` constants defined by `DateFormat`. They cause different details about the date to be presented. The argument `locale` is one of the static references defined by `Locale` (refer to Chapter 18 for details). If the style and/or locale is not specified, defaults are used.

One of the most commonly used methods in this class is `format( )`. It has several overloaded forms, one of which is shown here:

```java
final String format(Date d)
```

The argument is a `Date` object that is to be displayed. The method returns a string containing the formatted information.

The following listing illustrates how to format date information. It begins by creating a `Date` object. This captures the current date and time information. Then it outputs the date information by using different styles and locales.
Sample output from this program is shown here:

Japan: 11/01/01
Korea: 2011. 1. 1
United Kingdom: 01 January 2011
United States: Saturday, January 1, 2011

The `getTimeInstance()` method returns an instance of `DateFormat` that can format time information. It is available in these versions:

```
static final DateFormat getTimeInstance()
static final DateFormat getTimeInstance(int style)
static final DateFormat getTimeInstance(int style, Locale locale)
```

The argument `style` is one of the following values: DEFAULT, SHORT, MEDIUM, LONG, or FULL. These are
int constants defined by DateFormat. They cause different
details about the time to be presented. The argument
locale is one of the static references defined by Locale. If
the style and/or locale is not specified, defaults are used.

The following listing illustrates how to format time
information. It begins by creating a Date object. This
captures the current date and time information. Then it
outputs the time information by using different styles
and locales.

// Demonstrate time formats.
import java.text.*;
import java.util.*;
public class TimeFormatDemo {
    public static void main(String args[]) {
        Date date = new Date();
        DateFormat df;

        df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
        System.out.println("Japan: " + df.format(date));

        df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
        System.out.println("United Kingdom: " + df.format(date));

        df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
        System.out.println("Canada: " + df.format(date));
    }
}

Sample output from this program is shown here:

Japan: 20:25
United Kingdom: 20:25:14 CDT
Canada: 8:25:14 o'clock PM CDT

The DateFormat class also has a getTimeInstance(
) method that can format both date and time
information. You may wish to experiment with it on
SimpleDateFormat Class

SimpleDateFormat is a concrete subclass of DateFormat. It allows you to define your own formatting patterns that are used to display date and time information.

One of its constructors is shown here:

```java
SimpleDateFormat(String formatString)
```

The argument `formatString` describes how date and time information is displayed. An example of its use is given here:

```java
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

The symbols used in the formatting string determine the information that is displayed. Table 28-4 lists these symbols and gives a description of each.
In most cases, the number of times a symbol is
repeated determines how that data is presented. Text
information is displayed in an abbreviated form if the
pattern letter is repeated less than four times. Otherwise,
the unabbreviated form is used. For example, a *zzzz*
pattern can display Pacific Daylight Time, and a *zzz*
pattern can display PDT.
For numbers, the number of times a pattern letter is repeated determines how many digits are presented. For example, hh:mm:ss can present 01:51:15, but h:m:s displays the same time value as 1:51:15.

Finally, M or MM causes the month to be displayed as one or two digits. However, three or more repetitions of M cause the month to be displayed as a text string.

The following program shows how this class is used:

```java
// Demonstrate SimpleDateFormat.
import java.text.*;
import java.util.*;

public class SimpleDateFormatDemo {
    public static void main(String args[]) {
        Date date = new Date();
        SimpleDateFormat sdf;
        sdf = new SimpleDateFormat("hh:mm:ss");
        System.out.println(sdf.format(date));
        sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
        System.out.println(sdf.format(date));
        sdf = new SimpleDateFormat("E MMM dd yyyy");
        System.out.println(sdf.format(date));
    }
}
```

Sample output from this program is shown here:

```
12:46:49
01 Jan 2011 12:46:49 CST
Sat Jan 01 2011
```
PART III
Software Development Using Java

CHAPTER 29
Java Beans

CHAPTER 30
Introducing Swing

CHAPTER 31
Exploring Swing

CHAPTER 32
Servlets
This chapter provides an overview of Java Beans. Beans are important because they allow you to build complex systems from software components. These components may be provided by you or supplied by one or more different vendors. Java Beans defines an architecture that specifies how these building blocks can operate together.

To better understand the value of Beans, consider the following. Hardware designers have a wide variety of components that can be integrated together to construct a system. Resistors, capacitors, and inductors are examples of simple building blocks. Integrated circuits provide more advanced functionality. All of these different parts can be reused. It is not necessary or possible to rebuild these capabilities each time a new system is needed. Also, the same pieces can be used in different types of circuits. This is possible because the behavior of these components is understood and documented.

The software industry has also been seeking the benefits of reusability and interoperability of a component-based approach. To realize these benefits, a component architecture is needed that allows programs to be assembled from software building blocks, perhaps provided by different vendors. It must also be possible for a designer to select a component, understand its capabilities, and incorporate it into an application.
When a new version of a component becomes available, it should be easy to incorporate this functionality into existing code. Fortunately, Java Beans provides just such an architecture.

What Is a Java Bean?

A Java Bean is a software component that has been designed to be reusable in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as obtaining an inventory value, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface. A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block. Finally, a Bean may be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components.

Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally. However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

Advantages of Java Beans

The following list enumerates some of the benefits that
Java Bean technology provides for a component developer:

- A Bean obtains all the benefits of Java’s “write-once, run-anywhere” paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the runtime environment.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Introspection

At the core of Java Beans is introspection. This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the Java Beans technology could not operate.

There are two ways in which the developer of a Bean
can indicate which of its properties, events, and methods should be exposed. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class that extends the BeanInfo interface is provided that explicitly supplies this information. Both approaches are examined here.

Design Patterns for Properties
A property is a subset of a Bean’s state. The values assigned to the properties determine the behavior and appearance of that component. A property is set through a setter method. A property is obtained by a getter method. There are two types of properties: simple and indexed.

Simple Properties
A simple property has a single value. It can be identified by the following design patterns, where \( N \) is the name of the property and \( T \) is its type:

\[
\begin{align*}
\text{public } T & \ \text{getN()} \\
\text{public void } & \ \text{setN(T arg)}
\end{align*}
\]

A read/write property has both of these methods to access its values. A read-only property has only a get method. A write-only property has only a set method.

Here are three read/write simple properties along with their getter and setter methods:
private double depth, height, width;

public double getDepth() {
    return depth;
}
public void setDepth(double d) {
    depth = d;
}

public double getHeight() {
    return height;
}
public void setHeight(double h) {
    height = h;
}

public double getWidth() {
    return width;
}
public void setWidth(double w) {
    width = w;
}

Indexed Properties
An indexed property consists of multiple values. It can be identified by the following design patterns, where N is the name of the property and T is its type:

    public T getN(int index);
    public void setN(int index, T value);
    public T[] getN( );
    public void setN(T values[ ]); 

Here is an indexed property called data along with its getter and setter methods:

    private double data[ ];

    public double getData(int index) {
        return data[index];
    }
public void setData(int index, double value) {
    data[index] = value;
}

public double[] getData() {
    return data;
}

public void setData(double[] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
}

Design Patterns for Events

Beans use the delegation event model that was discussed earlier in this book. Beans can generate events and send them to other objects. These can be identified by the following design patterns, where T is the type of the event:

    public void addTListener(TListener eventListener)

public void addTListener(TListener eventListener)
    throws
        java.util.TooManyListenersException

public void removeTListener(TListener eventListener)

These methods are used to add or remove a listener for the specified event. The version of AddTListener() that does not throw an exception can be used to multicast an event, which means that more than one listener can register for the event notification. The version that throws TooManyListenersException unicasts the event, which means that the number of listeners is restricted to one. In either case, removeTListener() is used to remove the listener. For example, assuming an event interface
type called TemperatureListener, a Bean that monitors temperature might supply the following methods:

```java
public void addTemperatureListener(TemperatureListener tl) {
    ...
}
public void removeTemperatureListener(TemperatureListener tl) {
    ...
}
```

Methods and Design Patterns
Design patterns are not used for naming nonproperty methods. The introspection mechanism finds all of the public methods of a Bean. Protected and private methods are not presented.

Using the BeanInfo Interface
As the preceding discussion shows, design patterns implicitly determine what information is available to the user of a Bean. The BeanInfo interface enables you to explicitly control what information is available. The BeanInfo interface defines several methods, including these:

```java
PropertyDescriptor[ ] getPropertyDescriptors( )
EventSetDescriptor[ ] getEventSetDescriptors( )
MethodDescriptor[ ] getMethodDescriptors( )
```

They return arrays of objects that provide information about the properties, events, and methods of a Bean. The classes PropertyDescriptor, EventSetDescriptor, and MethodDescriptor are defined within the java.beans
package, and they describe the indicated elements. By implementing these methods, a developer can designate exactly what is presented to a user, bypassing introspection based on design patterns.

When creating a class that implements BeanInfo, you must call that class bnameBeanInfo, where bname is the name of the Bean. For example, if the Bean is called MyBean, then the information class must be called MyBeanBeanInfo.

To simplify the use of BeanInfo, JavaBeans supplies the SimpleBeanInfo class. It provides default implementations of the BeanInfo interface, including the three methods just shown. You can extend this class and override one or more of the methods to explicitly control what aspects of a Bean are exposed. If you don’t override a method, then design-pattern introspection will be used. For example, if you don’t override getPropertyDescriptors(), then design patterns are used to discover a Bean’s properties. You will see SimpleBeanInfo in action later in this chapter.

Bound and Constrained Properties

A Bean that has a bound property generates an event when the property is changed. The event is of type PropertyChangeEvent and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the PropertyChangeListener interface.
A Bean that has a constrained property generates an event when an attempt is made to change its value. It also generates an event of type PropertyChangeEvent. It too is sent to objects that previously registered an interest in receiving such notifications. However, those other objects have the ability to veto the proposed change by throwing a PropertyVetoException. This capability allows a Bean to operate differently according to its run-time environment. A class that handles this event must implement the VetoableChangeListener interface.

Persistence
Persistence is the ability to save the current state of a Bean, including the values of a Bean’s properties and instance variables, to nonvolatile storage and to retrieve them at a later time. The object serialization capabilities provided by the Java class libraries are used to provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the java.io.Serializable interface, which is simply a marker interface. Implementing java.io.Serializable makes serialization automatic. Your Bean need take no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements java.io.Serializable, then automatic serialization is obtained. There is one important restriction: any class that implements java.io.Serializable must supply a parameterless constructor.
When using automatic serialization, you can selectively prevent a field from being saved through the use of the transient keyword. Thus, data members of a Bean specified as transient will not be serialized.

If a Bean does not implement java.io.Serializable, you must provide serialization yourself, such as by implementing java.io.Externalizable. Otherwise, containers cannot save the configuration of your component.

Customizers

A Bean developer can provide a customizer that helps another developer configure the Bean. A customizer can provide a step-by-step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.
Table 29-1 The Interfaces in java.beans

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppletInitializer</td>
<td>Methods in this interface are used to initialize Beans that are also applets.</td>
</tr>
<tr>
<td>BeanInfo</td>
<td>This interface allows a designer to specify information about the properties, events, and methods of a Bean.</td>
</tr>
<tr>
<td>Customizer</td>
<td>This interface allows a designer to provide a graphical user interface through which a Bean may be configured.</td>
</tr>
<tr>
<td>DesignMode</td>
<td>Methods in this interface determine if a Bean is executing in design mode.</td>
</tr>
<tr>
<td>ExceptionListener</td>
<td>A method in this interface is invoked when an exception has occurred.</td>
</tr>
<tr>
<td>PropertyChangeListener</td>
<td>A method in this interface is invoked when a bound property is changed.</td>
</tr>
<tr>
<td>PropertyEditor</td>
<td>Objects that implement this interface allow designers to change and display property values.</td>
</tr>
<tr>
<td>VetoableChangeListener</td>
<td>A method in this interface is invoked when a constrained property is changed.</td>
</tr>
<tr>
<td>Visibility</td>
<td>Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.</td>
</tr>
</tbody>
</table>

The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the java.beans package. This section provides a brief overview of its contents. Table 29-1 lists the interfaces in java.beans and provides a brief description of their functionality. Table 29-2 lists the classes in java.beans.
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeanDescriptor</td>
<td>This class provides information about a Bean. It also allows you to associate a customizer with a Bean.</td>
</tr>
<tr>
<td>Beans</td>
<td>This class is used to obtain information about a Bean.</td>
</tr>
<tr>
<td>DefaultPersistenceDelegate</td>
<td>A concrete subclass of PersistenceDelegate.</td>
</tr>
<tr>
<td>Encoder</td>
<td>Encodes the state of a set of Beans. Can be used to write this information to a stream.</td>
</tr>
<tr>
<td>EventHandler</td>
<td>Supports dynamic event listener creation.</td>
</tr>
<tr>
<td>EventSetDescriptor</td>
<td>Instances of this class describe an event that can be generated by a Bean.</td>
</tr>
<tr>
<td>Expression</td>
<td>Encapsulates a call to a method that returns a result.</td>
</tr>
<tr>
<td>FeatureDescriptor</td>
<td>This is the superclass of the PropertyDescriptor, EventSetDescriptor, and MethodDescriptor classes.</td>
</tr>
<tr>
<td>IndexedPropertyChangeEvent</td>
<td>A subclass of PropertyChangeEvent that represents a change to an indexed property.</td>
</tr>
<tr>
<td>IndexedPropertyDescriptor</td>
<td>Instances of this class describe an indexed property of a Bean.</td>
</tr>
<tr>
<td>IntrospectionException</td>
<td>An exception of this type is generated if a problem occurs when analyzing a Bean.</td>
</tr>
<tr>
<td>Introspector</td>
<td>This class analyzes a Bean and constructs a BeanInfo object that describes the component.</td>
</tr>
<tr>
<td>MethodDescriptor</td>
<td>Instances of this class describe a method of a Bean.</td>
</tr>
<tr>
<td>ParameterDescriptor</td>
<td>Instances of this class describe a method parameter.</td>
</tr>
<tr>
<td>PersistenceDelegate</td>
<td>Handles the state information of an object.</td>
</tr>
</tbody>
</table>
Table 29-2 The Classes in java.beans

Although it is beyond the scope of this chapter to discuss all of the classes, four are of particular interest: Introspector, PropertyDescriptor, EventSetDescriptor, and MethodDescriptor. Each is briefly examined here.

Introspector

The Introspector class provides several static methods that support introspection. Of most interest is getBeanInfo(). This method returns a BeanInfo object that can be used to obtain information about the Bean. The getBeanInfo() method has several forms, including the one shown here:
static BeanInfo getBeanInfo(Class<?> bean) throws IntrospectionException

The returned object contains information about the Bean specified by bean.

PropertyDescriptor
The PropertyDescriptor class describes a Bean property. It supports several methods that manage and describe properties. For example, you can determine if a property is bound by calling isBound(). To determine if a property is constrained, call isConstrained(). You can obtain the name of a property by calling getName().

EventSetDescriptor
The EventSetDescriptor class represents a Bean event. It supports several methods that obtain the methods that a Bean uses to add or remove event listeners, and to otherwise manage events. For example, to obtain the method used to add listeners, call getAddListenerMethod(). To obtain the method used to remove listeners, call getRemoveListenerMethod(). To obtain the type of a listener, call getListenerType(). You can obtain the name of an event by calling getName().

MethodDescriptor
The MethodDescriptor class represents a Bean method. To obtain the name of the method, call getName(). You can obtain information about the method by calling
getMethod( ), shown here:

Method getMethod( )

An object of type Method that describes the method is returned.

A Bean Example
This chapter concludes with an example that illustrates various aspects of Bean programming, including introspection and using a BeanInfo class. It also makes use of the Introspector, PropertyDescriptor, and EventSetDescriptor classes. The example uses three classes. The first is a Bean called Colors, shown here:
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
    transient private Color color; // not persistent
    private boolean rectangular; // is persistent

    public Colors() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                change();
            }
        });
        rectangular = false;
        setSize(200, 100);
        change();
    }

    public boolean getRectangular() {
        return rectangular;
    }

    public void setRectangular(boolean flag) {
        this.rectangular = flag;
        repaint();
    }
}
public void change() {
    color = randomColor();
    repaint();
}

private Color randomColor() {
    int r = (int)(255*Math.random());
    int g = (int)(255*Math.random());
    int b = (int)(255*Math.random());
    return new Color(r, g, b);
}

public void paint(Graphics g) {
    Dimension d = getSize();
    int h = d.height;
    int w = d.width;
    g.setColor(color);
    if(rectangular) {
        g.fillRect(0, 0, w-1, h-1);
    } else {
        g.fillOval(0, 0, w-1, h-1);
    }
}

The Colors Bean displays a colored object within a frame. The color of the component is determined by the private Color variable color, and its shape is determined by the private boolean variable rectangular. The constructor defines an anonymous inner class that extends MouseAdapter and overrides its mousePressed( ) method. The change( ) method is invoked in response to mouse presses. It selects a random color and then repaints the component. The getRectangular( ) and setRectangular( ) methods provide access to the one
property of this Bean. The change( ) method calls randomColor( ) to choose a color and then calls repaint( ) to make the change visible. Notice that the paint( ) method uses the rectangular and color variables to determine how to present the Bean.

The next class is ColorsBeanInfo. It is a subclass of SimpleBeanInfo that provides explicit information about Colors. It overrides getPropertyDescriptors( ) in order to designate which properties are presented to a Bean user. In this case, the only property exposed is rectangular. The method creates and returns a PropertyDescriptor object for the rectangular property. The PropertyDescriptor constructor that is used is shown here:

```java
PropertyDescriptor(String property, Class<?> beanCls)
    throws IntrospectionException
```

Here, the first argument is the name of the property, and the second argument is the class of the Bean.

```java
// A Bean information class.
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor rectangular = new PropertyDescriptor("rectangular", Colors.class);
            PropertyDescriptor pd[] = {rectangular};
            return pd;
        }
        catch(Exception e) {
            System.out.println("Exception caught. " + e);
        }
    }
}
```
The final class is called `IntrospectorDemo`. It uses introspection to display the properties and events that are available within the `Colors` Bean.

```java
// Show properties and events.
import java.awt.*;
import java.beans.*;

public class IntrospectorDemo {
    public static void main(String args[]) {
        try {
            Class<?> c = Class.forName("Colors");
            BeanInfo beanInfo = Introspector.getBeanInfo(c);

            System.out.println("Properties:");
            PropertyDescriptor propertyDescriptor[] =
                beanInfo.getPropertyDescriptors();
            for(int i = 0; i < propertyDescriptor.length; i++) {
                System.out.println("\t" + propertyDescriptor[i].getName());
            }

            System.out.println("Events:");
            EventSetDescriptor eventSetDescriptor[] =
                beanInfo.getEventSetDescriptors();
            for(int i = 0; i < eventSetDescriptor.length; i++) {
                System.out.println("\t" + eventSetDescriptor[i].getName());
            }
        } catch(Exception e) {
            System.out.println("Exception caught. " + e);
        }
    }
}
```

The output from this program is the following:

Properties:
- rectangular

Events:
- mouseWheel
- mouse
Notice two things in the output. First, because ColorsBeanInfo overrides getPropertyDescriptors() such that the only property returned is rectangular, only the rectangular property is displayed. However, because getEventSetDescriptors() is not overridden by ColorsBeanInfo, design-pattern introspection is used, and all events are found, including those in Colors’ superclass, Canvas. Remember, if you don’t override one of the “get” methods defined by SimpleBeanInfo, then the default, design-pattern introspection is used. To observe the difference that ColorsBeanInfo makes, erase its class file and then run IntrospectorDemo again. This time it will report more properties.
CHAPTER 30
Introducing Swing

In Part II, you saw how to build user interfaces with the AWT classes. Although the AWT is still a crucial part of Java, its component set is no longer widely used to create graphical user interfaces. Today, most programmers use Swing for this purpose. Swing is a set of classes that provides more powerful and flexible GUI components than does the AWT. Simply put, Swing provides the look and feel of the modern Java GUI.

Coverage of Swing is divided between two chapters. This chapter introduces Swing. It begins by describing Swing's core concepts. It then shows the general form of a Swing program, including both applications and applets. It concludes by explaining how painting is accomplished in Swing. The following chapter presents several commonly used Swing components. It is important to understand that the number of classes and interfaces in the Swing packages is quite large, and they can't all be covered in this book. (In fact, full coverage of Swing requires an entire book of its own.) However, these two chapters will give you a basic understanding of this important topic.

NOTE

The Origins of Swing
Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java’s original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as heavyweight.

The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always rectangular and opaque.

Not long after Java’s original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach
was needed. The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing Is Built on the AWT
Before moving on, it is necessary to make one important point: although Swing eliminates a number of the limitations inherent in the AWT, Swing does not replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing. (The AWT is covered in Chapters 24 and 25. Event handling is described in Chapter 23.)

Two Key Swing Features
As just explained, Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing. Each is examined here.

Swing Components Are Lightweight
With very few exceptions, Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Swing supports a pluggable look and feel (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply “plugged in.” Once this is done, all components are automatically rendered using that style.
Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

Java 7 provides look-and-feels, such as metal, Motif, and Nimbus, that are available to all Swing users. The metal look and feel is also called the Java look and feel. It is platform-independent and available in all Java execution environments. It is also the default look and feel. Windows environments also have access to the Windows and Windows Classic look and feel. This book uses the default Java look and feel (metal) because it is platform independent.

The MVC Connection
In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component
No matter what architecture is used to implement a component, it must implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: Model-View-Controller, or MVC for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the model corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The view determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The controller determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user’s choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a
modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate. For this reason, Swing’s approach is called either the Model-Delegate architecture or the Separable Model architecture. Therefore, although Swing’s component architecture is based on MVC, it does not use a classical implementation of it.

Swing’s pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.

To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined by interfaces. For example, the model for a button is defined by the ButtonModel interface. UI delegates are classes that inherit ComponentUI. For example, the UI delegate for a button is ButtonUI. Normally, your programs will not interact directly with the UI delegate.

Components and Containers
A Swing GUI consists of two key items: components and containers. However, this distinction is mostly conceptual
because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a component is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a containment hierarchy, at the top of which must be a top-level container.

Let's look a bit more closely at components and containers.

Components

In general, Swing components are derived from the JComponent class. (The only exceptions to this are the four top-level containers, described in the next section.) JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component.

All of Swing’s components are represented by classes defined within the package javax.swing. The following
table shows the class names for Swing components (including those used as containers).

<table>
<thead>
<tr>
<th>JApplet</th>
<th>JButton</th>
<th>JCheckBox</th>
<th>JCheckBoxMenuItem</th>
</tr>
</thead>
<tbody>
<tr>
<td>JColorChooser</td>
<td>JComboBox</td>
<td>JComponent</td>
<td>JDesktopPane</td>
</tr>
<tr>
<td>JDialog</td>
<td>JEditorPane</td>
<td>JFileChooser</td>
<td>JFormattedTextField</td>
</tr>
<tr>
<td>JFrame</td>
<td>JFrameInternal</td>
<td>JLabel</td>
<td>JLayer</td>
</tr>
<tr>
<td>JLayeredPane</td>
<td>JList</td>
<td>JMenu</td>
<td>JMenuBar</td>
</tr>
<tr>
<td>JMenuItem</td>
<td>JOptionPane</td>
<td>JPanel</td>
<td>JPasswordField</td>
</tr>
<tr>
<td>JPopupMenu</td>
<td>JProgressBar</td>
<td>JRadioButton</td>
<td>JRadioButtonMenuItem</td>
</tr>
<tr>
<td>JRootPane</td>
<td>JScrollPane</td>
<td>JSeparator</td>
<td></td>
</tr>
<tr>
<td>JSlider</td>
<td>JSpinner</td>
<td>JSplitPane</td>
<td>JTabbedPane</td>
</tr>
<tr>
<td>JTable</td>
<td>JTextArea</td>
<td>JTextField</td>
<td>JTextPane</td>
</tr>
<tr>
<td>JViewport</td>
<td>JToolBar</td>
<td>JToolTip</td>
<td>JTree</td>
</tr>
<tr>
<td>JWindow</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice that all component classes begin with the letter J. For example, the class for a label is JLabel; the class for a push button is JButton; and the class for a scroll bar is JScrollPane.

Containers

Swing defines two types of containers. The first are top-level containers: JFrame, JApplet, JWindow, and JDialog. These containers do not inherit JComponent. They do, however, inherit the AWT classes Component and Container. Unlike Swing’s other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container.
Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is JFrame. The one used for applets is JApplet.

The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit JComponent. An example of a lightweight container is JPanel, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as JPanel to create subgroups of related controls that are contained within an outer container.

The Top-Level Container Panes

Each top-level container defines a set of panes. At the top of the hierarchy is an instance of JRootPane. JRootPane is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar. The panes that comprise the root pane are called the glass pane, the content pane, and the layered pane.

The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of JPanel. The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint
over any other component, for example. In most cases, you won’t need to use the glass pane directly, but it is there if you need it.

The layered pane is an instance of JLayeredPane. The layered pane allows components to be given a depth value. This value determines which component overlays another. (Thus, the layered pane lets you specify a Z-order for a component, although this is not something that you will usually need to do.) The layered pane holds the content pane and the (optional) menu bar.

Although the glass pane and the layered panes are integral to the operation of a top-level container and serve important purposes, much of what they provide occurs behind the scene. The pane with which your application will interact the most is the content pane, because this is the pane to which you will add visual components. In other words, when you add a component, such as a button, to a top-level container, you will add it to the content pane. By default, the content pane is an opaque instance of JPanel.

The Swing Packages
Swing is a very large subsystem and makes use of many packages. At the time of this writing, these are the packages defined by Swing.
The main package is javax.swing. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

A Simple Swing Application

Swing programs differ from both the console-based programs and the AWT-based programs shown earlier in this book. For example, they use a different set of components and a different container hierarchy than does the AWT. Swing programs also have special requirements that relate to threading. The best way to understand the structure of a Swing program is to work through an example. There are two types of Java programs in which Swing is typically used. The first is a desktop application. The second is the applet. This section shows how to create a Swing application. The creation of a Swing applet is described later in this chapter.

Although quite short, the following program shows one way to write a Swing application. In the process, it demonstrates several key features of Swing. It uses two Swing components: JFrame and JLabel. JFrame is the
top-level container that is commonly used for Swing applications. JLabel is the Swing component that creates a label, which is a component that displays information. The label is Swing’s simplest component because it is passive. That is, a label does not respond to user input. It just displays output. The program uses a JFrame container to hold an instance of a JLabel. The label displays a short text message.
Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

```
javac SwingDemo.java
```

To run the program, use this command line:
When the program is run, it will produce the window shown in Figure 30-1.

Because the SwingDemo program illustrates several core Swing concepts, we will examine it carefully, line by line. The program begins by importing javax.swing. As mentioned, this package contains the components and models defined by Swing. For example, javax.swing defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing.

Next, the program declares the SwingDemo class and a constructor for that class. The constructor is where most of the action of the program occurs. It begins by creating a JFrame, using this line of code:

```java
JFrame jfrm = new JFrame("A Simple Swing Application");
```

![A Simple Swing Application window](image)

Figure 30-1 The window produced by the SwingDemo program

This creates a container called jfrm that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The
Next, the window is sized using this statement:

```java
jfrm.setSize(275, 100);
```

The `setSize()` method (which is inherited by JFrame from the AWT class Component) sets the dimensions of the window, which are specified in pixels. Its general form is shown here:

```java
void setSize(int width, int height)
```

In this example, the width of the window is set to 275 and the height is set to 100.

By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated. While this default behavior is useful in some situations, it is not what is needed for most applications. Instead, you will usually want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call `setDefaultCloseOperation()`, as the program does:

```java
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

After this call executes, closing the window causes the entire application to terminate. The general form of `setDefaultCloseOperation()` is shown here:

```java
void setDefaultCloseOperation(int what)
```
The value passed in what determines what happens when the window is closed. There are several other options in addition to JFrame.EXIT_ON_CLOSE. They are shown here:

- DISPOSE_ON_CLOSE
- HIDE_ON_CLOSE
- DO NOTHING_ON_CLOSE

Their names reflect their actions. These constants are declared in WindowConstants, which is an interface declared in javax.swing that is implemented by JFrame.

The next line of code creates a Swing JLabel component:

```java
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
```

JLabel is the simplest and easiest-to-use component because it does not accept user input. It simply displays information, which can consist of text, an icon, or a combination of the two. The label created by the program contains only text, which is passed to its constructor.

The next line of code adds the label to the content pane of the frame:

```java
jfrm.add(jlab);
```

As explained earlier, all top-level containers have a content pane in which components are stored. Thus, to
add a component to a frame, you must add it to the frame’s content pane. This is accomplished by calling add( ) on the JFrame reference (jfrm in this case). The general form of add( ) is shown here:

Component add(Component comp)

The add( ) method is inherited by JFrame from the AWT class Container.

By default, the content pane associated with a JFrame uses border layout. The version of add( ) just shown adds the label to the center location. Other versions of add( ) enable you to specify one of the border regions. When a component is added to the center, its size is adjusted automatically to fit the size of the center.

Before continuing, an important historical point needs to be made. Prior to JDK 5, when adding a component to the content pane, you could not invoke the add( ) method directly on a JFrame instance. Instead, you needed to call add( ) on the content pane of the JFrame object. The content pane can be obtained by calling getContentPane( ) on a JFrame instance. The getContentPane( ) method is shown here:

Container getContentPane( )

It returns a Container reference to the content pane. The add( ) method was then called on that reference to add a component to a content pane. Thus, in the past, you had to use the following statement to add jlab to jfrm:

```java
jfrm.getContentPane().add(jlab);
```
jfrm.getContentPane().add(jlab); // old-style

Here, getContentPane( ) first obtains a reference to content pane, and then add( ) adds the component to the container linked to this pane. This same procedure was also required to invoke remove( ) to remove a component and setLayout( ) to set the layout manager for the content pane. You will see explicit calls to getContentPane( ) frequently throughout pre-5.0 code. Today, the use of getContentPane( ) is no longer necessary. You can simply call add( ), remove( ), and setLayout( ) directly on JFrame because these methods have been changed so that they operate on the content pane automatically.

The last statement in the SwingDemo constructor causes the window to become visible:

jfrm.setVisible(true);

The setVisible( ) method is inherited from the AWT Component class. If its argument is true, the window will be displayed. Otherwise, it will be hidden. By default, a JFrame is invisible, so setVisible(true) must be called to show it.

Inside main( ), a SwingDemo object is created, which causes the window and the label to be displayed. Notice that the SwingDemo constructor is invoked using these lines of code:

SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        ...
    }
});
This sequence causes a SwingDemo object to be created on the event dispatching thread rather than on the main thread of the application. Here’s why. In general, Swing programs are event-driven. For example, when a user interacts with a component, an event is generated. An event is passed to the application by calling an event handler defined by the application. However, the handler is executed on the event dispatching thread provided by Swing and not on the main thread of the application. Thus, although event handlers are defined by your program, they are called on a thread that was not created by your program.

To avoid problems (including the potential for deadlock), all Swing GUI components must be created and updated from the event dispatching thread, not the main thread of the application. However, main( ) is executed on the main thread. Thus, main( ) cannot directly instantiate a SwingDemo object. Instead, it must create a Runnable object that executes on the event dispatching thread and have this object create the GUI.

To enable the GUI code to be created on the event dispatching thread, you must use one of two methods that are defined by the SwingUtilities class. These methods are invokeLater( ) and invokeAndWait( ). They are shown here:
Here, obj is a Runnable object that will have its run( ) method called by the event dispatching thread. The difference between the two methods is that invokeLater( ) returns immediately, but invokeAndWait( ) waits until obj.run( ) returns. You can use one of these methods to call a method that constructs the GUI for your Swing application, or whenever you need to modify the state of the GUI from code not executed by the event dispatching thread. You will normally want to use invokeLater( ), as the preceding program does. However, when constructing the initial GUI for an applet, you will need to use invokeAndWait( ).

Event Handling
The preceding example showed the basic form of a Swing program, but it left out one important part: event handling. Because JLabel does not take input from the user, it does not generate events, so no event handling was needed. However, the other Swing components do respond to user input and the events generated by those interactions need to be handled. Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any
Swing-based application.

The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called the delegation event model, and it is described in Chapter 23. In many cases, Swing uses the same events as does the AWT, and these events are packaged in java.awt.event. Events specific to Swing are stored in javax.swing.event.

Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button. Sample output is shown in Figure 30-2.

![Figure 30-2](image)

Figure 30-2 Output from the EventDemo program
// Handle an event in a Swing program.

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

class EventDemo {

    JLabel jlab;

    EventDemo() {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("An Event Example");

        // Specify FlowLayout for the layout manager.
        jfrm.setLayout(new FlowLayout());

        // Give the frame an initial size.
        jfrm.setSize(220, 90);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
First, notice that the program now imports both the java.awt and java.awt.event packages. The java.awt
package is needed because it contains the FlowLayout class, which supports the standard flow layout manager used to lay out components in a frame. (See Chapter 25 for coverage of layout managers.) The java.awt.event package is needed because it defines the ActionListener interface and the ActionEvent class.

The EventDemo constructor begins by creating a JFrame called jfrm. It then sets the layout manager for the content pane of jfrm to FlowLayout. Recall that, by default, the content pane uses BorderLayout as its layout manager. However, for this example, FlowLayout is more convenient. Notice that FlowLayout is assigned using this statement:

```java
jfrm.setLayout(new FlowLayout());
```

As explained, in the past you had to explicitly call `getContentPane()` to set the layout manager for the content pane. This requirement was removed as of JDK 5.

After setting the size and default close operation, EventDemo() creates two push buttons, as shown here:

```java
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
```

The first button will contain the text “Alpha” and the second will contain the text “Beta”. Swing push buttons are instances of JButton. JButton supplies several constructors. The one used here is
The `msg` parameter specifies the string that will be displayed inside the button.

When a push button is pressed, it generates an `ActionEvent`. Thus, `JButton` provides the `addActionListener( )` method, which is used to add an action listener. (`JButton` also provides `removeActionListener( )` to remove a listener, but this method is not used by the program.) As explained in Chapter 23, the `ActionListener` interface defines only one method: `actionPerformed( )`. It is shown again here for your convenience:

```java
void actionPerformed(ActionEvent ae)
```

This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.

Next, event listeners for the button’s action events are added by the code shown here:

```java
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
    }
});

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
    }
});
```
Here, anonymous inner classes are used to provide the event handlers for the two buttons. Each time a button is pressed, the string displayed in jlab is changed to reflect which button was pressed.

Next, the buttons are added to the content pane of jfrm:

```java
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

Finally, jlab is added to the content pane and window is made visible. When you run the program, each time you press a button, a message is displayed in the label that indicates which button was pressed.

One last point: Remember that all event handlers, such as `actionPerformed()`, are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If your application needs to do something time consuming as the result of an event, it must use a separate thread.

Create a Swing Applet
The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference: A Swing applet extends JApplet rather than Applet. JApplet is derived from Applet. Thus, JApplet includes all of the functionality found in Applet and adds support for
Swing. JApplet is a top-level Swing container, which means that it is not derived from JComponent. Because JApplet is a top-level container, it includes the various panes described earlier. This means that all components are added to JApplet’s content pane in the same way that components are added to JFrame’s content pane.

Swing applets use the same four lifecycle methods as described in Chapter 22: init(), start(), stop(), and destroy(). Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the paint() method. (Painting in Swing is described later in this chapter.)

One other point: All interaction with components in a Swing applet must take place on the event dispatching thread, as described in the previous section. This threading issue applies to all Swing programs.

Here is an example of a Swing applet. It provides the same functionality as the previous application, but does so in applet form. Figure 30-3 shows the program when executed by appletviewer.
Figure 30-3 Output from the example Swing applet

// A simple Swing-based applet

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

/**
 * This HTML can be used to launch the applet:
 *
<applet code="MySwingApplet" width=220 height=90>
</applet>
*/

public class MySwingApplet extends JApplet {
    JButton jbtnAlpha;
    JButton jbtnBeta;
    JLabel jlab;

    // Initialize the applet.
    public void init() {
        try {
        SwingUtilities.invokeLater(new Runnable () {
            public void run() {
                makeGUI(); // initialize the GUI
            }
        });
    }
}
} catch (Exception exc) {
    System.out.println("Can't create because of " + exc);
}

// This applet does not need to override start(), stop(),
// or destroy().

// Set up and initialize the GUI.
private void makeGUI() {

    // Set the applet to use flow layout.
    setLayout(new FlowLayout());

    // Make two buttons.
    jbtnAlpha = new JButton("Alpha");
    jbtnBeta = new JButton("Beta");

    // Add action listener for Alpha.
    jbtnAlpha.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            jlab.setText("Alpha was pressed.");
        }
    });
There are two important things to notice about this applet. First, MySwingApplet extends JApplet. As explained, all Swing-based applets extend JApplet rather than Applet. Second, the init() method initializes the Swing components on the event dispatching thread by setting up a call to makeGUI(). Notice that this is accomplished through the use of invokeAndWait() rather than invokeLater(). Applets must use invokeAndWait() because the init() method must not return until the entire initialization process has been completed. In essence, the start() method cannot be called until after initialization, which means that the GUI must be fully constructed.

Inside makeGUI(), the two buttons and label are
created, and the action listeners are added to the buttons. Finally, the components are added to the content pane. Although this example is quite simple, this same general approach must be used when building any Swing GUI that will be used by an applet.

Painting in Swing
Although the Swing component set is quite powerful, you are not limited to using it because Swing also lets you write directly into the display area of a frame, panel, or one of Swing’s other components, such as JLabel. Although many (perhaps most) uses of Swing will not involve drawing directly to the surface of a component, it is available for those applications that need this capability. To write output directly to the surface of a component, you will use one or more drawing methods defined by the AWT, such as drawLine() or drawRect(). Thus, most of the techniques and methods described in Chapter 24 also apply to Swing. However, there are also some very important differences, and the process is discussed in detail in this section.

Painting Fundamentals
Swing’s approach to painting is built on the original AWT-based mechanism, but Swing’s implementation offers more finally grained control. Before examining the specifics of Swing-based painting, it is useful to review the AWT-based mechanism that underlies it.

The AWT class Component defines a method called
paint( ) that is used to draw output directly to the surface of a component. For the most part, paint( ) is not called by your program. (In fact, only in the most unusual cases should it ever be called by your program.) Rather, paint( ) is called by the run-time system whenever a component must be rendered. This situation can occur for several reasons. For example, the window in which the component is displayed can be overwritten by another window and then uncovered. Or, the window might be minimized and then restored. The paint( ) method is also called when a program begins running. When writing AWT-based code, an application will override paint( ) when it needs to write output directly to the surface of the component.

Because JComponent inherits Component, all Swing’s lightweight components inherit the paint( ) method. However, you will not override it to paint directly to the surface of a component. The reason is that Swing uses a bit more sophisticated approach to painting that involves three distinct methods: paintComponent( ), paintBorder( ), and paintChildren( ). These methods paint the indicated portion of a component and divide the painting process into its three distinct, logical actions. In a lightweight component, the original AWT method paint( ) simply executes calls to these methods, in the order just shown.

To paint to the surface of a Swing component, you will create a subclass of the component and then
override its `paintComponent()` method. This is the method that paints the interior of the component. You will not normally override the other two painting methods. When overriding `paintComponent()`, the first thing you must do is call `super.paintComponent()`, so that the superclass portion of the painting process takes place. (The only time this is not required is when you are taking complete, manual control over how a component is displayed.) After that, write the output that you want to display. The `paintComponent()` method is shown here:

```java
protected void paintComponent(Graphics g)
```

The parameter `g` is the graphics context to which output is written.

To cause a component to be painted under program control, call `repaint()`. It works in Swing just as it does for the AWT. The `repaint()` method is defined by `Component`. Calling it causes the system to call `paint()` as soon as it is possible to do so. Because painting is a time-consuming operation, this mechanism allows the run-time system to defer painting momentarily until some higher-priority task has completed, for example. Of course, in Swing the call to `paint()` results in a call to `paintComponent()`. Therefore, to output to the surface of a component, your program will store the output until `paintComponent()` is called. Inside the overridden `paintComponent()`, you will draw the stored output.
Compute the Paintable Area

When drawing to the surface of a component, you must be careful to restrict your output to the area that is inside the border. Although Swing automatically clips any output that will exceed the boundaries of a component, it is still possible to paint into the border, which will then get overwritten when the border is drawn. To avoid this, you must compute the paintable area of the component. This is the area defined by the current size of the component minus the space used by the border. Therefore, before you paint to a component, you must obtain the width of the border and then adjust your drawing accordingly.

To obtain the border width, call `getInsets()`, shown here:

```java
class Insets {
    int top;
    int bottom;
    int left;
    int right;
}
```

To obtain the paintable area of the component, you can subtract the border dimensions from the component's size. This ensures that your drawing is confined to the area inside the border.
These values are then used to compute the drawing area given the width and the height of the component. You can obtain the width and height of the component by calling `getWidth()` and `getHeight()` on the component. They are shown here:

```java
int getWidth()

int getHeight()
```

By subtracting the value of the insets, you can compute the usable width and height of the component.

A Paint Example
Here is a program that puts into action the preceding discussion. It creates a class called `PaintPanel` that extends `JPanel`. The program then uses an object of that class to display lines whose endpoints have been generated randomly. Sample output is shown in Figure 10-4.

![Sample output from the PaintPanel program](image)

Figure 30-4 Sample output from the PaintPanel program
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// This class extends JPanel. It overrides
// the paintComponent() method so that random
// lines are plotted in the panel.
class PaintPanel extends JPanel {
    Insets ins; // holds the panel’s insets
    Random rand; // used to generate random numbers

    // Construct a panel.
    PaintPanel() {
        // Put a border around the panel.
        setBorder(
            BorderFactory.createLineBorder(Color.RED, 5));

        rand = new Random();
    }
}
// Override the paintComponent() method.
protected void paintComponent(Graphics g) {
    // Always call the superclass method first.
    super.paintComponent(g);

    int x, y, x2, y2;

    // Get the height and width of the component.
    int height = getHeight();
    int width = getWidth();

    // Get the insets.
    ins = getInsets();

    // Draw ten lines whose endpoints are randomly generated.
    for(int i=0; i < 10; i++) {
        // Obtain random coordinates that define
        // the endpoints of each line.
        x = rand.nextInt(width-ins.left);
        y = rand.nextInt(height-ins.bottom);
        x2 = rand.nextInt(width-ins.left);
        y2 = rand.nextInt(height-ins.bottom);
// Draw the line.
g.drawLine(x, y, x2, y2);

// Demonstrate painting directly onto a panel.
class PaintDemo {

    JLabel jlab;
    PaintPanel pp;

    PaintDemo() {

        // Create a new JFrame container.
        JFrame jfrm = new JFrame("Paint Demo");

        // Give the frame an initial size.
        jfrm.setSize(200, 150);

        // Terminate the program when the user closes the application.
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create the panel that will be painted.
        pp = new PaintPanel();

        // Add the panel to the content pane. Because the default
        // border layout is used, the panel will automatically be
        // sized to fit the center region.
        jfrm.add(pp);

        // Display the frame.
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // Create the frame on the event dispatching thread.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new PaintDemo();
            }
        });
    }
}
Let’s examine this program closely. The PaintPanel class extends JPanel. JPanel is one of Swing’s lightweight containers, which means that it is a component that can be added to the content pane of a JFrame. To handle painting, PaintPanel overrides the paintComponent( ) method. This enables PaintPanel to write directly to the surface of the component when painting takes place. The size of the panel is not specified because the program uses the default border layout and the panel is added to the center. This results in the panel being sized to fill the center. If you change the size of the window, the size of the panel will be adjusted accordingly.

Notice that the constructor also specifies a 5-pixel wide, red border. This is accomplished by setting the border by using the setBorder( ) method, shown here:

```java
void setBorder(Border border)
```

Border is the Swing interface that encapsulates a border. You can obtain a border by calling one of the factory methods defined by the BorderFactory class. The one used in the program is createLineBorder( ), which creates a simple line border. It is shown here:

```java
static Border createLineBorder(Color clr, int width)
```

Here, clr specifies the color of the border and width specifies its width in pixels.

Inside the override of paintComponent( ), notice that it first calls super.paintComponent( ). As explained, this
is necessary to ensure that the component is properly
drawn. Next the width and height of the panel are
obtained along with the insets. These values are used to
ensure the lines lie within the drawing area of the panel.
The drawing area is the overall width and height of a
component less the border width. The computations are
designed to work with differently sized PaintPanels and
borders. To prove this, try changing the size of the
window. The lines will still all lie within the borders of
the panel.

The PaintDemo class creates a PaintPanel and then
adds the panel to the content pane. When the
application is first displayed, the overridden
paintComponent( ) method is called, and the lines are
drawn. Each time you resize or hide and restore the
window, a new set of lines are drawn. In all cases, the
lines fall within the paintable area.
CHAPTER 31
Exploring Swing

The previous chapter described several of the core concepts relating to Swing and showed the general form of both a Swing application and a Swing applet. This chapter continues the discussion of Swing by presenting an overview of several Swing components, such as buttons, check boxes, trees, and tables. The Swing components provide rich functionality and allow a high level of customization. Because of space limitations, it is not possible to describe all of their features and attributes. Rather, the purpose of this overview is to give you a feel for the capabilities of the Swing component set.

The Swing component classes described in this chapter are shown here:

<table>
<thead>
<tr>
<th>JButton</th>
<th>JCheckBox</th>
<th>JComboBox</th>
<th>JLabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>JList</td>
<td>JRadioButton</td>
<td>JScrollPane</td>
<td>JTabbedPane</td>
</tr>
<tr>
<td>JTable</td>
<td>JTextField</td>
<td>JToggleButton</td>
<td>JTree</td>
</tr>
</tbody>
</table>

These components are all lightweight, which means that they are all derived from JComponent.

Also discussed is the ButtonGroup class, which encapsulates a mutually exclusive set of Swing buttons, and ImageIcon, which encapsulates a graphics image. Both are defined by Swing and packaged in javax.swing.

One other point: The Swing components are demonstrated in applets because the code for an applet
is more compact than it is for a desktop application. However, the same techniques apply to both applets and applications.

**JLabel and ImageIcon**

JLabel is Swing’s easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at JLabel a bit more closely. JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. JLabel defines several constructors. Here are three of them:

- JLabel(Icon icon)
- JLabel(String str)
- JLabel(String str, Icon icon, int align)

Here, str and icon are the text and icon used for the label. The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: LEFT, RIGHT, CENTER, LEADING, or TRAILING. These constants are defined in the SwingConstants interface, along with several others used by the Swing classes.

Notice that icons are specified by objects of type Icon, which is an interface defined by Swing. The easiest way to obtain an icon is to use the ImageIcon class. ImageIcon implements Icon and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument
to the Icon parameter of JLabel’s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the ImageIcon constructor used by the example in this section:

```
ImageIcon(String filename)
```

It obtains the image in the file named filename.

The icon and text associated with the label can be obtained by the following methods:

```
Icon getIcon( )
String getText( )
```

The icon and text associated with a label can be set by these methods:

```
void setIcon(Icon icon)
void setText(String str)
```

Here, icon and str are the icon and text, respectively. Therefore, using setText( ) it is possible to change the text inside a label during program execution.

The following applet illustrates how to create and display a label containing both an icon and a string. It begins by creating an ImageIcon object for the file france.gif, which depicts the flag for France. This is used as the second argument to the JLabel constructor. The first and last arguments for the JLabel constructor are the
Demonstrate JLabel and ImageIcon.
import java.awt.*;
import javax.swing.*;

/**
   <applet code="JLabelDemo" width=250 height=150>
    </applet>
*/

public class JLabelDemo extends JApplet {

    public void init() {
        try {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    makeGUI();
                }
            });
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }

    private void makeGUI() {

        // Create an icon.
        ImageIcon ii = new ImageIcon("france.gif");

        // Create a label.
        JLabel jl = new JLabel("France", ii, JLabel.CENTER);

        // Add the label to the content pane.
        add(jl);
    }
}
Output from the label example is shown here:

![Applet Viewer: JLabelDemo](image)

**JTextField**

JTextField is the simplest Swing text component. It is also probably its most widely used text component. JTextField allows you to edit one line of text. It is derived from JTextComponent, which provides the basic functionality common to Swing text components. JTextField uses the Document interface for its model.

Three of JTextField’s constructors are shown here:

1. `JTextField(int cols)`
2. `JTextField(String str, int cols)`
3. `JTextField(String str)`

Here, str is the string to be initially presented, and cols is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.

JTextField generates events in response to user
interaction. For example, an ActionEvent is fired when the user presses ENTER. A CaretEvent is fired each time the caret (i.e., the cursor) changes position. (CaretEvent is packaged in javax.swing.event.) Other events are also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call getText().

The following example illustrates JTextField. It creates a JTextField and adds it to the content pane. When the user presses ENTER, an action event is generated. This is handled by displaying the text in the status window.

```java
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
   <applet code="JTextFieldDemo" width=300 height=50>
   </applet>
*/

public class JTextFieldDemo extends JApplet {
    JTextField jtf;

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                });
            } catch (Exception exc) {
```
The Swing Buttons

Swing defines four types of buttons: JButton, JToggleButton, JCheckBox, and JRadioButton. All are subclasses of the AbstractButton class, which extends JComponent. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon
can be used as a rollover icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```java
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Here, di, pi, si, and ri are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

```java
String getText()
void setText(String str)
```

Here, str is the text to be associated with the button.

The model used by all buttons is defined by the ButtonModel interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

**JButton**

The JButton class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. JButton allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:
Here, `str` and `icon` are the string and icon used for the button.

When the button is pressed, an `ActionEvent` is generated. Using the `ActionEvent` object passed to the `actionPerformed()` method of the registered `ActionListener`, you can obtain the action command string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling `setActionCommand()` on the button. You can obtain the action command by calling `getActionCommand()` on the event object. It is declared like this:

```java
String getActionCommand()
```

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the label.
Demonstrate an icon-based JButton.

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

/*
   <applet code="JButtonDemo" width=250 height=450>
</applet>
*/

public class JButtonDemo extends JApplet
implements ActionListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                });
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
```
private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Add buttons to content pane.
    ImageIcon france = new ImageIcon("france.gif");
    JButton jb = new JButton(france);
    jb.setActionCommand("France");
    jb.addActionListener(this);
    add(jb);

    ImageIcon germany = new ImageIcon("germany.gif");
    jb = new JButton(germany);
    jb.setActionCommand("Germany");
    jb.addActionListener(this);
    add(jb);

    ImageIcon italy = new ImageIcon("italy.gif");
    jb = new JButton(italy);
    jb.setActionCommand("Italy");
    jb.addActionListener(this);
    add(jb);

    ImageIcon japan = new ImageIcon("japan.gif");
    jb = new JButton(japan);
    jb.setActionCommand("Japan");
    jb.addActionListener(this);
    add(jb);

    // Create and add the label to content pane.
    JLabel jlab = new JLabel("Choose a Flag");
    add(jlab);
}

// Handle button events.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
A useful variation on the push button is called a toggle button. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.
Toggle buttons are objects of the JToggleButton class. JToggleButton implements AbstractButton. In addition to creating standard toggle buttons, JToggleButton is a superclass for two other Swing components that also represent two-state controls. These are JCheckBox and JRadioButton, which are described later in this chapter. Thus, JToggleButton defines the basic functionality of all two-state components.

JToggleButton defines several constructors. The one used by the example in this section is shown here:

JToggleButton(String str)

This creates a toggle button that contains the text passed in str. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

JToggleButton uses a model defined by a nested class called JToggleButton.ToggleButtonModel. Normally, you won’t need to interact directly with the model to use a standard toggle button.

Like JButton, JToggleButton generates an action event each time it is pressed. Unlike JButton, however, JToggleButton also generates an item event. This event is used by those components that support the concept of selection. When a JToggleButton is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the ItemListener interface. Recall from Chapter 23, that each
time an item event is generated, it is passed to the
itemStateChanged( ) method defined by ItemListener.
Inside itemStateChanged( ), the getItem( ) method can be
called on the ItemEvent object to obtain a reference to
the JToggleButton instance that generated the event. It is
shown here:

    Object getItem( )

A reference to the button is returned. You will need to
cast this reference to JToggleButton.

The easiest way to determine a toggle button’s state is
by calling the isSelected( ) method (inherited from
AbstractButton) on the button that generated the event. It
is shown here:

    boolean isSelected( )

It returns true if the button is selected and false
otherwise.

Here is an example that uses a toggle button. Notice
how the item listener works. It simply calls isSelected( )
to determine the button’s state.
Demonstrate JToggleButton.

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

public class JToggleButtonDemo extends JApplet {

    JLabel jlab;
    JToggleButton jtbn;

    public void init() {  
        try {
            SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                makeGUI();
            };
        };
```
The output from the toggle button example is shown here:
Check Boxes

The JCheckBox class provides the functionality of a check box. Its immediate superclass is JToggleButton, which provides support for two-state buttons, as just described. JCheckBox defines several constructors. The one used here is

    JCheckBox(String str)

It creates a check box that has the text specified by str as a label. Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a check box, an ItemEvent is generated. You can obtain a reference to the JCheckBox that generated the event by calling getItem() on the ItemEvent passed to the itemStateChanged() method defined by ItemListener. The easiest way to determine the selected state of a check box is to call isSelected() on the JCheckBox instance.

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an ItemEvent is generated. Inside the itemStateChanged() method, getItem() is called to

![Applet Viewer: JToggleButtonDemo](image)

Applet started.
obtain a reference to the JCheckBox object that generated the event. Next, a call to isSelected() determines if the box was selected or cleared. The getText() method gets the text for that check box and uses it to set the text inside the label.

```java
// Demonstrate JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/*
   <applet code="JCheckBoxDemo" width=270 height=50>
   </applet>
*/

public class JCheckBoxDemo extends JApplet
implements ItemListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                });
        }
    }
```
private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Add check boxes to the content pane.
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener(this);
    add(cb);
    cb = new JCheckBox("C++");
    cb.addItemListener(this);
    add(cb);
    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    add(cb);
Radio Buttons

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the JRadioButton class, which extends JToggleButton. JRadioButton provides several constructors. The one used in the example is shown here:

JRadioButton(String str)
Here, \texttt{str} is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the \texttt{ButtonGroup} class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

\begin{verbatim}
void add(AbstractButton ab)
\end{verbatim}

Here, \texttt{ab} is a reference to the button to be added to the group.

A \texttt{JRadioButton} generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means that you will normally implement the \texttt{ActionListener} interface. Recall that the only method defined by \texttt{ActionListener} is \texttt{actionPerformed( )}. Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling \texttt{getActionCommand( )}. By default, the action command is the same as the button label, but you can set the action
command to something else by calling setActionCommand( ) on the radio button. Second, you can call getSource( ) on the ActionEvent object and check that reference against the buttons. Finally, you can simply check each radio button to find out which one is currently selected by calling isSelected( ) on each button.

Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by actionPerformed( ). Within that handler, the getActionCommand( ) method gets the text that is associated with the radio button and uses it to set the text within a label.
// Demonstrate JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/*
   <applet code="JRadioButtonDemo" width=300 height=50>
   </applet>
*/

public class JRadioButtonDemo extends JApplet
    implements ActionListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                });
        }
    }
catch (Exception exc) {
    System.out.println("Can't create because of " + exc);
}

private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Create radio buttons and add them to content pane.
    JRadioButton bl = new JRadioButton("A");
    bl.addActionListener(this);
    add(bl);

    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    add(b2);

    JRadioButton b3 = new JRadioButton("C");
    b3.addActionListener(this);
    add(b3);

    // Define a button group.
    ButtonGroup bg = new ButtonGroup();
    bg.add(bl);
    bg.add(b2);
    bg.add(b3);

    // Create a label and add it to the content pane.
    jlab = new JLabel("Select One");
    add(jlab);
}

// Handle button selection.
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
}
JTabbedPane

JTabbedPane encapsulates a tabbed pane. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront. Tabbed panes are very common in the modern GUI, and you have no doubt used them many times. Given the complex nature of a tabbed pane, they are surprisingly easy to create and use.

JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides. JTabbedPane uses the SingleSelectionModel model.

Tabs are added by calling addTab( ). Here is one of its forms:

```java
void addTab(String name, Component comp)
```

Here, name is the name for the tab, and comp is the component that should be added to the tab. Often, the component added to a tab is a JPanel that contains a group of related components. This technique allows a tab to hold a set of components.
The general procedure to use a tabbed pane is outlined here:

1. Create an instance of JTabbedPane.
2. Add each tab by calling addTab().
3. Add the tabbed pane to the content pane.

The following example illustrates a tabbed pane. The first tab is titled “Cities” and contains four buttons. Each button displays the name of a city. The second tab is titled “Colors” and contains three check boxes. Each check box displays the name of a color. The third tab is titled “Flavors” and contains one combo box. This enables the user to select one of three flavors.
import javax.swing.*;
/*
   <applet code="JTabbedPaneDemo" width=400 height=100>
   </applet>
*/

public class JTabbedPaneDemo extends JApplet {

    public void init() { 
        try {
            SwingUtilities.invokeLater(
                    new Runnable() {
                        public void run() {
                            makeGUI();
                        }
                    });
        } catch (Exception exc) {
            System.out.println("Can't create because of "+ exc);
        }
    }
}
private void makeGUI() {

    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    add(jtp);
}

// Make the panels that will be added to the tabbed pane.
class CitiesPanel extends JPanel {

    public CitiesPanel() {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
    }
}
Output from the tabbed pane example is shown in the following three illustrations:
JScrollPane

JScrollPane is a lightweight container that automatically handles the scrolling of another component. The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a JPanel. In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because JScrollPane automates scrolling, it usually eliminates the need to manage individual scroll bars.

The viewable area of a scroll pane is called the viewport. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the
visible portion of the component being scrolled. The scroll bars scroll the component through the viewport. In its default behavior, a JScrollPane will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors. The one used in this chapter is shown here:

```java
JScrollPane(Component comp)
```

The component to be scrolled is specified by comp. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

1. Create the component to be scrolled.
2. Create an instance of JScrollPane, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a JPanel object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.
// Demonstrate JScrollPane.
import java.awt.*;
import javax.swing.*;
/*
 <applet code="JScrollPaneDemo" width=300 height=250>
 </applet>
*/

public class JScrollPaneDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
      } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
      }
    }
  }
}
private void makeGUI() {
    // Add 400 buttons to a panel.
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20, 20));
    int b = 0;

    for(int i = 0; i < 20; i++) {
        for(int j = 0; j < 20; j++) {
            jp.add(new JButton("Button "+ b));
            ++b;
        }
    }

    // Create the scroll pane.
    JScrollPane jsp = new JScrollPane(jp);

    // Add the scroll pane to the content pane.
    // Because the default border layout is used,
    // the scroll pane will be added to the center.
    add(jsp, BorderLayout.CENTER);
}

Output from the scroll pane example is shown here:
In Swing, the basic list class is called JList. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. JList is so widely used in Java that it is highly unlikely that you have not seen one before.

In the past, the items in a JList were represented as Object references. However, with the release of JDK 7, JList was made generic and is now declared like this:

```java
class JList<E>
```

Here, E represents the type of the items in the list.

JList provides several constructors. The one used here is

```java
JList(E[] items)
```

This creates a JList that contains the items in the array specified by items.

JList is based on two models. The first isListModel. This interface defines how access to the list data is achieved. The second model is the ListSelectionModel interface, which defines methods that determine what list item or items are selected.

Although a JList will work properly by itself, most of the time you will wrap a JList inside a JScrollPane. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the
A JList generates a ListSelectionEvent when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing ListSelectionListener. This listener specifies only one method, called valueChanged( ), which is shown here:

```java
void valueChanged(ListSelectionEvent le)
```

Here, le is a reference to the object that generated the event. Although ListSelectionEvent does provide some methods of its own, normally you will interrogate the JList object itself to determine what has occurred. Both ListSelectionEvent and ListSelectionListener are packaged in javax.swing.event.

By default, a JList allows the user to select multiple ranges of items within the list, but you can change this behavior by calling setSelectionMode( ), which is defined by JList. It is shown here:

```java
void setSelectionMode(int mode)
```

Here, mode specifies the selection mode. It must be one of these values defined by ListSelectionModel:

- SINGLE_SELECTION
- SINGLE_INTERVAL_SELECTION
The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It’s just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling `getSelectedIndex()`:

```
int getSelectedIndex()
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, \(-1\) is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling `getSelectedValue()`:

```
E getSelectedValue()
```

It returns a reference to the first selected value. If no value has been selected, it returns `null`.

The following applet demonstrates a simple `JList`, which holds a list of cities. Each time a city is selected in the list, a `ListSelectionEvent` is generated, which is
handled by the valueChanged( ) method defined by ListSelectionListener. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

```java
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/*
  <applet code="JListDemo" width=200 height=120>
  </applet>
*/

public class JListDemo extends JApplet {
    JList<String> jlst;
    JLabel jlab;
    JScrollPane jscrlp;

    // Create an array of cities.

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                });
        }
    }
```
private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Create a JList.
    jlst = new JList<String>(Cities);

    // Set the list selection mode to single selection.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Add the list to a scroll pane.
    jscrlp = new JScrollPane(jlst);

    // Set the preferred size of the scroll pane.
    jscrlp.setPreferredSize(new Dimension(120, 90));
// Make a label that displays the selection.
JLabel jlab = new JLabel("Choose a City");

// Add selection listener for the list.
jlist.addListSelectionListener(new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent le)
    {
        // Get the index of the changed item.
        int idx = jlist.getSelectedIndex();
        
        // Display selection, if item was selected.
        if(idx != -1)
            jlab.setText("Current selection: " + Cities[idx]);
        else // Otherwise, reprompt.
            jlab.setText("Choose a City");
    
    
});

// Add the list and label to the content pane.
add(jscroll);
add(jlab);

Output from the list example is shown here:

JComboBox
Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class. A combo box normally displays one entry, but it
will also display a drop-down list that allows a user to select a different entry. You can also create a combo box that lets the user enter a selection into the text field.

In the past, the items in a JComboBox were represented as Object references. However, with the release of JDK 7, JComboBox was made generic and is now declared like this:

```java
class JComboBox<E>
```

Here, E represents the type of the items in the combo box.

The JComboBox constructor used by the example is shown here:

```java
JComboBox(E[] items)
```

Here, items is an array that initializes the combo box. Other constructors are available.

JComboBox uses the ComboBoxModel. Mutable combo boxes (those whose entries can be changed) use the MutableComboBoxModel.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the addItem() method, shown here:

```java
void addItem(E obj)
```

Here, obj is the object to be added to the combo box.
This method must be used only with mutable combo boxes.

JComboBox generates an action event when the user selects an item from the list. JComboBox also generates an item event when the state of selection changes, which occurs when an item is selected or deselected. Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.

One way to obtain the item selected in the list is to call `getSelectedItem()` on the combo box. It is shown here:

```
Object getSelectedItem()
```

You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for “France,” “Germany,” “Italy,” and “Japan.” When a country is selected, an icon-based label is updated to display the flag for that country. You can see how little code is required to use this powerful component.
// Demonstrate JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
   <applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/

public class JComboBoxDemo extends JApplet {
    JLabel jlab;
    ImageIcon france, germany, italy, japan;
    JComboBox<String> jcb;

    String flags[] = { "France", "Germany", "Italy", "Japan" };

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
Trees
A tree is a component that presents a hierarchical view of data. The user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the JTree class. A sampling of its constructors is shown here:
JTree(Object obj [ ])
JTree(Vector<?> v)
JTree(TreeNode tn)

In the first form, the tree is constructed from the elements in the array obj. The second form constructs the tree from the elements of vector v. In the third form, the tree whose root node is specified by tn specifies the tree.

Although JTree is packaged in javax.swing, its support classes and interfaces are packaged in javax.swing.tree. This is because the number of classes and interfaces needed to support JTree is quite large.

JTree relies on two models: TreeModel and TreeSelectionModel. A JTree generates a variety of events, but three relate specifically to trees: TreeExpansionEvent, TreeSelectionEvent, and TreeModelEvent. TreeExpansionEvent events occur when a node is expanded or collapsed. A TreeSelectionEvent is generated when the user selects or deselects a node within the tree. A TreeModelEvent is fired when the data or structure of the tree changes. The listeners for these events are TreeExpansionListener, TreeSelectionListener, and TreeModelListener, respectively. The tree event classes and listener interfaces are packaged in javax.swing.event.

The event handled by the sample program shown in this section is TreeSelectionEvent. To listen for this event, implement TreeSelectionListener. It defines only one
method, called valueChanged(), which receives the TreeSelectionEvent object. You can obtain the path to the selected object by calling getPath(), shown here, on the event object:

```
TreePath getPath()
```

It returns a TreePath object that describes the path to the changed node. The TreePath class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the toString() method is used. It returns a string that describes the path.

The TreeNode interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The MutableTreeNode interface extends TreeNode. It declares methods that can insert and remove child nodes or change the parent node.

The DefaultMutableTreeNode class implements the MutableTreeNode interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, obj is the object to be enclosed in this tree node. The new tree node doesn’t have a parent or children.

To create a hierarchy of tree nodes, the add() method
DefaultMutableTreeNode can be used. Its signature is shown here:

```java
void add(MutableTreeNode child)
```

Here, child is a mutable tree node that is to be added as a child to the current node.

JTree does not provide any scrolling capabilities of its own. Instead, a JTree is typically placed within a JScrollPane. This way, a large tree can be scrolled through a smaller viewport.

Here are the steps to follow to use a tree:

1. Create an instance of JTree.
2. Create a JScrollPane and specify the tree as the object to be scrolled.
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create a tree and handle selections. The program creates a DefaultMutableTreeNode instance labeled “Options”. This is the top node of the tree hierarchy. Additional tree nodes are then created, and the add( ) method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the JTree constructor. The tree is then provided as the argument to the JScrollPane constructor. This scroll pane is then added to the content pane. Next, a label is created and added to the content pane. The tree selection is
displayed in this label. To receive selection events from the tree, a TreeSelectionListener is registered for the tree. Inside the valueChanged( ) method, the path to the current selection is obtained and displayed.

```java
// Demonstrate JTree.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;

public class JTreeDemo extends JApplet {
    JTree tree;
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(new Runnable() {
                public void run() {
                    makeGUI();
                }
            });
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```
private void makeGUI() {

    // Create top node of tree.
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

    // Create subtree of "A".
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
    a.add(a2);

    // Create subtree of "B"
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
    b.add(b3);

    // Create the tree.
    tree = new JTree(top);

    // Add the tree to a scroll pane.
    JScrollPane jsp = new JScrollPane(tree);

    // Add the scroll pane to the content pane.
    add(jsp);

    // Add the label to the content pane.
    JLabel jlab = new JLabel();
    add(jlab, BorderLayout.SOUTH);

    // Handle tree selection events.
    tree.addTreeSelectionListener(new TreeSelectionListener() {
        public void valueChanged(TreeSelectionEvent tse) {
            jlab.setText("Selection is " + tse.getPath());
        }
    });
}
The string presented in the text field describes the path from the top tree node to the selected node.

**JTable**

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell. JTable is a sophisticated component that offers many more options and features than can be discussed here. (It is perhaps Swing’s most complicated component.) However, in its default configuration, JTable still offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format. The brief overview presented here will give you a general understanding of this powerful component.

Like JTree, JTable has many classes and interfaces
At its core, JTable is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. JTable does not provide any scrolling capabilities of its own. Instead, you will normally wrap a JTable inside a JScrollPane.

JTable supplies several constructors. The one used here is

```
JTable(Object data[ ][ ], Object colHeads[ ])
```

Here, `data` is a two-dimensional array of the information to be presented, and `colHeads` is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the `TableModel` interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by `TableColumnModel`. JTable is defined in terms of columns, and it is `TableColumnModel` that specifies the characteristics of a column. These two models are packaged in `javax.swing.table`. The third model determines how items are selected, and it is specified by
the ListSelectionModel, which was described when JList was discussed.

A JTable can generate several different events. The two most fundamental to a table’s operation are ListSelectionEvent and TableModelEvent. A ListSelectionEvent is generated when the user selects something in the table. By default, JTable allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A TableModelEvent is fired when that table’s data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use JTable to display data (as the following example does), then you don’t need to handle any events.

Here are the steps required to set up a simple JTable that can be used to display data:

1. Create an instance of JTable.
2. Create a JScrollPane object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called colHeads is created for the column headings. A
A two-dimensional array of strings called data is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the JTable constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the data array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is data in this case.

```java
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;

public class JTableDemo extends JApplet {

public void init() {
    try {
        SwingUtilities.invokeLater(
            new Runnable() {
                public void run() {
                    makeGUI();
                }
            });
    }
}
```
catch (Exception exc) {
    System.out.println("Can't create because of " + exc);
}

private void makeGUI() {

    // Initialize column headings.
    String[] colHeads = { "Name", "Extension", "ID#" };

    // Initialize data.
    Object[][] data = {
        { "Gail", "4567", "865" },
        { "Ken", "7566", "555" },
        { "Viviane", "5634", "587" },
        { "Melanie", "7345", "922" },
        { "Anne", "1237", "333" },
        { "John", "5656", "314" },
        { "Matt", "5672", "217" },
        { "Claire", "6741", "444" },
        { "Erwin", "9023", "519" },
        { "Ellen", "1134", "532" },
        { "Jennifer", "5689", "112" },
        { "Ed", "9030", "133" },
        { "Helen", "6751", "145" }
    };

    // Create the table.
    JTable table = new JTable(data, colHeads);

    // Add the table to a scroll pane.
    JScrollPane jsp = new JScrollPane(table);

    // Add the scroll pane to the content pane.
    add(jsp);
}
Continuing Your Exploration of Swing

Swing defines a very large GUI toolkit. It has many more features that you will want to explore on your own. For example, Swing provides toolbars, tooltips, and progress bars. It also provides a complete menu subsystem. Swing’s pluggable look and feel lets you substitute another appearance and behavior for an element. You can define your own models for the various components, and you can change the way that cells are edited and rendered when working with tables and trees. The best way to become familiar with Swing’s capabilities is to experiment with it.
This chapter presents an overview of servlets. Servlets are small programs that execute on the server side of a web connection. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. The topic of servlets is quite large, and it is beyond the scope of this chapter to cover it all. Instead, we will focus on the core concepts, interfaces, and classes, and develop several examples.

Background
In order to understand the advantages of servlets, you must have a basic understanding of how web browsers and servers cooperate to provide content to a user. Consider a request for a static web page. A user enters a Uniform Resource Locator (URL) into a browser. The browser generates an HTTP request to the appropriate web server. The web server maps this request to a specific file. That file is returned in an HTTP response to the browser. The HTTP header in the response indicates the type of the content. The Multipurpose Internet Mail Extensions (MIME) are used for this purpose. For example, ordinary ASCII text has a MIME type of text/plain. The Hypertext Markup Language (HTML) source code of a web page has a MIME type of text/html.

Now consider dynamic content. Assume that an online store uses a database to store information about its business. This would include items for sale, prices,
availability, orders, and so forth. It wishes to make this information accessible to customers via web pages. The contents of those web pages must be dynamically generated to reflect the latest information in the database.

In the early days of the Web, a server could dynamically construct a page by creating a separate process to handle each client request. The process would open connections to one or more databases in order to obtain the necessary information. It communicated with the web server via an interface known as the Common Gateway Interface (CGI). CGI allowed the separate process to read data from the HTTP request and write data to the HTTP response. A variety of different languages were used to build CGI programs. These included C, C++, and Perl.

However, CGI suffered serious performance problems. It was expensive in terms of processor and memory resources to create a separate process for each client request. It was also expensive to open and close database connections for each client request. In addition, the CGI programs were not platform-independent. Therefore, other techniques were introduced. Among these are servlets.

Servlets offer several advantages in comparison with CGI. First, performance is significantly better. Servlets execute within the address space of a web server. It is not necessary to create a separate process to handle each client request. Second, servlets are platform-independent because they are written in Java. Third, the Java security
The Life Cycle of a Servlet
Three methods are central to the life cycle of a servlet. These are `init()`, `service()`, and `destroy()`. They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, assume that a user enters a Uniform Resource Locator (URL) to a web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.

Second, this HTTP request is received by the web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the server invokes the `init()` method of the servlet. This method is invoked only when the servlet is first loaded into memory. It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the server invokes the `service()` method of the servlet. This method is called to process the HTTP request. You will see that it is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client.
The servlet remains in the server’s address space and is available to process any other HTTP requests received from clients. The `service()` method is called for each HTTP request.

Finally, the server may decide to unload the servlet from its memory. The algorithms by which this determination is made are specific to each server. The server calls the `destroy()` method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

Servlet Development Options

To create servlets, you will need access to a servlet container/server. Two popular ones are Glassfish and Tomcat. Glassfish is from Oracle and is provided by the Java EE SDK. It is supported by NetBeans. Tomcat is an open-source product maintained by the Apache Software Foundation. It can also be used by NetBeans. Both Tomcat and Glassfish can also be used with other IDEs, such as Eclipse. The examples and descriptions in this chapter use Tomcat for reasons that will soon be apparent.

Although IDEs such as NetBeans and Eclipse are very useful and can streamline the creation of servlets, they are not used in this chapter. The way you develop and deploy servlets differs among IDEs, and it is simply not possible for this book to address each environment. Furthermore, many readers will be using the command-line tools rather than an IDE. Therefore, if you are using...
an IDE, you must refer to the instructions for that environment for information concerning the development and deployment of servlets. For this reason, the instructions given here and elsewhere in this chapter assume that only the command-line tools are employed. Thus, they will work for nearly any reader.

Tomcat is used in this chapter because, in the opinion of this author, it makes it relatively easy to run the example servlets using only command-line tools and a text editor. It is also widely available in various programming environments. Furthermore, since only command-line tools are used, you don’t need to download and install an IDE just to experiment with servlets. Understand, however, that even if you are developing in an environment that uses Glassfish, the concepts presented here still apply. It is just that the mechanics of preparing a servlet for testing will be slightly different.

REMEMBER

The instructions for developing and deploying servlets in this chapter are based on Tomcat and use only command-line tools. If you are using an IDE and different servlet container/server, consult the documentation for your environment.

Using Tomcat

Tomcat contains the class libraries, documentation, and run-time support that you will need to create and test servlets. At the time of this writing, several versions of Tomcat are available for use, including 5.5.x, 6.0.x, and
7.0.x. All will work with the examples in this chapter. However, the instructions that follow use 7.0.4, which supports servlet specification 3.0. You can download Tomcat from tomcat.apache.org. Tomcat versions 6.0.x and 7.0.x support both 32-bit and 64-bit Windows. You should choose a version appropriate to your environment.

The examples in this chapter assume a 64-bit Windows environment. Assuming that a 64-bit version of Tomcat 7.0.4 was unpacked from the root directly, the default location is

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\n```

This is the location assumed by the examples in this book. If you load Tomcat in a different location (or use a different version of Tomcat), you will need to make appropriate changes to the examples. You may need to set the environmental variable JAVA_HOME to the top-level directory in which the Java Development Kit is installed.

**NOTE**

All of the directories shown in this section assume Tomcat 7.0.4. If you install a different version of Tomcat, then you will need to adjust the directory names and paths to match those used by the version you installed.

Once installed, you start Tomcat by selecting startup.bat from the bin directory directly under the apache-tomcat-7.0.4 directory. To stop Tomcat, execute shutdown.bat, also in the bin directory.
The classes and interfaces needed to build servlets are contained in servlet-api.jar, which is in the following directory:

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\lib
```

To make servlet-api.jar accessible, update your CLASSPATH environment variable so that it includes

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\lib\servlet-api.jar
```

Alternatively, you can specify this file when you compile the servlets. For example, the following command compiles the first servlet example:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\lib\servlet-api.jar"
```

Once you have compiled a servlet, you must enable Tomcat to find it. For our purposes, this means putting it into a directory under Tomcat’s webapps directory and entering its name into a web.xml file. To keep things simple, the examples in this chapter use the directory and web.xml file that Tomcat supplies for its own example servlets. This way, you won’t have to create any files or directories just to experiment with the sample servlets. Here is the procedure that you will follow.

First, copy the servlet’s class file into the following directory:

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\webapps\examples\WEB-INF\classes
```

Next, add the servlet’s name and mapping to the web.xml file in the following directory:

```
C:\apache-tomcat-7.0.4-windows-x64\apache-tomcat-7.0.4\webapps\examples\WEB-INF\classes
```
For instance, assuming the first example, called HelloServlet, you will add the following lines in the section that defines the servlets:

```xml
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Next, you will add the following lines to the section that defines the servlet mappings:

```xml
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Follow this same general procedure for all of the examples.

A Simple Servlet
To become familiar with the key servlet concepts, we will begin by building and testing a simple servlet. The basic steps are the following:

1. Create and compile the servlet source code. Then, copy the servlet’s class file to the proper directory, and add the servlet’s name and mappings to the proper web.xml file.

2. Start Tomcat.

3. Start a web browser and request the servlet.

Let us examine each of these steps in detail.

Create and Compile the Servlet Source Code
To begin, create a file named HelloServlet.java that
Let's look closely at this program. First, note that it imports the javax.servlet package. This package contains the classes and interfaces required to build servlets. You will learn more about these later in this chapter. Next, the program defines HelloServlet as a subclass of GenericServlet. The GenericServlet class provides functionality that simplifies the creation of a servlet. For example, it provides versions of init( ) and destroy( ), which may be used as is. You need supply only the service( ) method.

Inside HelloServlet, the service( ) method (which is inherited from GenericServlet) is overridden. This method handles requests from a client. Notice that the first argument is a ServletRequest object. This enables the servlet to read data that is provided via the client request. The second argument is a ServletResponse object. This enables the servlet to formulate a response for the client.

The call to setContentType( ) establishes the MIME type of the HTTP response. In this program, the MIME
type is text/html. This indicates that the browser should interpret the content as HTML source code.

Next, the `getWriter()` method obtains a PrintWriter. Anything written to this stream is sent to the client as part of the HTTP response. Then `println()` is used to write some simple HTML source code as the HTTP response.

Compile this source code and place the HelloServlet.class file in the proper Tomcat directory as described in the previous section. Also, add HelloServlet to the web.xml file, as described earlier.

Start Tomcat
Start Tomcat as explained earlier. Tomcat must be running before you try to execute a servlet.

Start a Web Browser and Request the Servlet
Start a web browser and enter the URL shown here:

http://localhost:8080/examples/servlets/servlet/HelloServlet

Alternatively, you may enter the URL shown here:

http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet

This can be done because 127.0.0.1 is defined as the IP address of the local machine.

You will observe the output of the servlet in the browser display area. It will contain the string Hello! in bold type.

The Servlet API
Two packages contain the classes and interfaces that are required to build the servlets described in this chapter.
These are `javax.servlet` and `javax.servlet.http`. They constitute the Servlet API. Keep in mind that these packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they are provided by Tomcat. They are also provided by Java EE.

The Servlet API has been in a process of ongoing development and enhancement. The current servlet specification is version 3.0, and that is the one used in this book. However, because changes happen fast in the world of Java, you will want to check for any additions or alterations. This chapter discusses the core of the Servlet API, which will be available to most readers.

### The `javax.servlet` Package

The `javax.servlet` package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes the core interfaces that are provided in this package. The most significant of these is `Servlet`. All servlets must implement this interface or extend a class that implements the interface. The `ServletRequest` and `ServletResponse` interfaces are also very important.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servlet</td>
<td>Declares life cycle methods for a servlet.</td>
</tr>
<tr>
<td>ServletConfig</td>
<td>Allows servlets to get initialization parameters.</td>
</tr>
<tr>
<td>ServletContext</td>
<td>Enables servlets to log events and access information about their environment.</td>
</tr>
<tr>
<td>ServletRequest</td>
<td>Used to read data from a client request.</td>
</tr>
<tr>
<td>ServletResponse</td>
<td>Used to write data to a client response.</td>
</tr>
</tbody>
</table>

The following table summarizes the core classes that are provided in the `javax.servlet` package:
Let us examine these interfaces and classes in more detail.

The Servlet Interface

All servlets must implement the Servlet interface. It declares the init( ), service( ), and destroy( ) methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters. The methods defined by Servlet are shown in Table 32-1.

The init( ), service( ), and destroy( ) methods are the life cycle methods of the servlet. These are invoked by the server. The getServletConfig( ) method is called by the servlet to obtain initialization parameters. A servlet developer overrides the getServletInfo( ) method to provide a string with useful information (for example, author, version, date, copyright). This method is also invoked by the server.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenericServlet</td>
<td>Implements the Servlet and ServletConfig interfaces.</td>
</tr>
<tr>
<td>ServletInputStream</td>
<td>Provides an input stream for reading requests from a client.</td>
</tr>
<tr>
<td>ServletOutputStream</td>
<td>Provides an output stream for writing responses to a client.</td>
</tr>
<tr>
<td>ServletException</td>
<td>Indicates a servlet error occurred.</td>
</tr>
<tr>
<td>UnavailableException</td>
<td>Indicates a servlet is unavailable.</td>
</tr>
</tbody>
</table>
Table 32-1 The Methods Defined by Servlet

The ServletConfig Interface

The ServletConfig interface allows a servlet to obtain configuration data when it is loaded. The methods declared by this interface are summarized here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void destroy()</td>
<td>Called when the servlet is unloaded.</td>
</tr>
<tr>
<td>ServletConfig getServletConfig()</td>
<td>Returns a <code>ServletConfig</code> object that contains any initialization parameters.</td>
</tr>
<tr>
<td>String getServletInfo()</td>
<td>Returns a string describing the servlet.</td>
</tr>
<tr>
<td>void init(ServletConfig sc) throws ServletException</td>
<td>Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from <code>sc</code>. A <code>ServletException</code> should be thrown if the servlet cannot be initialized.</td>
</tr>
<tr>
<td>void service(ServletRequest req, ServletResponse res) throws ServletException, IOException</td>
<td>Called to process a request from a client. The request from the client can be read from <code>req</code>. The response to the client can be written to <code>res</code>. An exception is generated if a servlet or IO problem occurs.</td>
</tr>
</tbody>
</table>

The ServletContext Interface

The ServletContext interface enables servlets to obtain information about their environment. Several of its methods are summarized in Table 32-2.

The ServletRequest Interface

The ServletRequest interface enables a servlet to obtain information about a client request. Several of its methods are summarized in Table 32-3.

The ServletResponse Interface
The `ServletResponse` interface enables a servlet to formulate a response for a client. Several of its methods are summarized in Table 32-4.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getAttribute(String attr)</code></td>
<td>Returns the value of the server attribute named <code>attr</code>.</td>
</tr>
<tr>
<td><code>getMimeType(String file)</code></td>
<td>Returns the MIME type of <code>file</code>.</td>
</tr>
<tr>
<td><code>getRealPath(String vpath)</code></td>
<td>Returns the real path that corresponds to the virtual path <code>vpath</code>.</td>
</tr>
<tr>
<td><code>getServerInfo()</code></td>
<td>Returns information about the server.</td>
</tr>
<tr>
<td><code>log(String s)</code></td>
<td>Writes <code>s</code> to the servlet log.</td>
</tr>
<tr>
<td><code>log(String s, Throwable e)</code></td>
<td>Writes <code>s</code> and the stack trace for <code>e</code> to the servlet log.</td>
</tr>
<tr>
<td><code>setAttribute(String attr, Object val)</code></td>
<td>Sets the attribute specified by <code>attr</code> to the value passed in <code>val</code>.</td>
</tr>
</tbody>
</table>

Table 32-2 Various Methods Defined by `ServletContext`
The GenericServlet Class

The GenericServlet class provides implementations of the basic life cycle methods for a servlet. GenericServlet implements the Servlet and ServletConfig interfaces. In addition, a method to append a string to the server log file is available. The signatures of this method are shown here:

```java
void log(String s)
void log(String s, Throwable e)
```

Here, s is the string to be appended to the log, and e is an exception that occurred.

The ServletInputStream Class

The ServletInputStream class extends InputStream. It is implemented by the servlet container and provides an input stream that a servlet developer can use to read the data from a client request. It defines the default constructor. In addition, a method is provided to read

---

**Table 32-3 Various Methods Defined by HttpServletRequest**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getCharacterEncoding()</td>
<td>Returns the character encoding for the response.</td>
</tr>
<tr>
<td>ServletOutputStream getOutputStream() throws IOException</td>
<td>Returns a ServletOutputStream that can be used to write binary data to the response. An IllegalStateException is thrown if getOutputStream() has already been invoked for this request.</td>
</tr>
<tr>
<td>PrintWriter getWriter() throws IOException</td>
<td>Returns a PrintWriter that can be used to write character data to the response. An IllegalStateException is thrown if getOutputStream() has already been invoked for this request.</td>
</tr>
<tr>
<td>void setContentLength(int size)</td>
<td>Sets the content length for the response to size.</td>
</tr>
<tr>
<td>void setContentType(String type)</td>
<td>Sets the content type for the response to type.</td>
</tr>
</tbody>
</table>

---

**Table 32-4 Various Methods Defined by HttpServletResponse**
bytes from the stream. It is shown here:

```java
int readLine(byte[] buffer, int offset, int size) throws IOException
```

Here, `buffer` is the array into which `size` bytes are placed starting at `offset`. The method returns the actual number of bytes read or −1 if an end-of-stream condition is encountered.

The ServletOutputStream Class

The `ServletOutputStream` class extends `OutputStream`. It is implemented by the servlet container and provides an output stream that a servlet developer can use to write data to a client response. A default constructor is defined. It also defines the `print()` and `println()` methods, which output data to the stream.

The Servlet Exception Classes

`javax.servlet` defines two exceptions. The first is `ServletException`, which indicates that a servlet problem has occurred. The second is `UnavailableException`, which extends `ServletException`. It indicates that a servlet is unavailable.

Reading Servlet Parameters

The `ServletRequest` interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in `PostParameters.html`, and a servlet is defined in `PostParametersServlet.java`. 

The HTML source code for PostParameters.html is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```html
<html>
<body>
<center>
<form name="Form1"
   method="post"
   action="http://localhost:8080/examples/servlets/servlet/PostParametersServlet">
<table>
<tr>
   <td><B>Employee</td>
   <td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
   <td><B>Phone</td>
   <td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

The source code for PostParametersServlet.java is shown in the following listing. The service( ) method is overridden to process client requests. The getParameterNames( ) method returns an enumeration of the parameter names. These are processed in a loop. You can see that the parameter name and value are output to the client. The parameter value is obtained via the getParameter( ) method.

```java
import java.io.*;
import java.util.*;
import javax.servlet.*;
```
public class PostParametersServlet
extends GenericServlet {

    public void service(ServletRequest request,
            ServletResponse response)
        throws ServletException, IOException {

        // Get print writer.
        PrintWriter pw = response.getWriter();

        // Get enumeration of parameter names.
        Enumeration e = request.getParameterNames();

        // Display parameter names and values.
        while(e.hasMoreElements()) {
            String pname = (String)e.nextElement();
            pw.print(pname + " = ");
            String pvalue = request.getParameter(pname);
            pw.println(pvalue);
        }
        pw.close();
    }
}

Compile the servlet. Next, copy it to the appropriate
directory, and update the web.xml file, as previously
described. Then, perform these steps to test this example:

1. Start Tomcat (if it is not already running).
2. Display the web page in a browser.
3. Enter an employee name and phone number in
   the text fields.
4. Submit the web page.

After following these steps, the browser will display a
response that is dynamically generated by the servlet.

The javax.servlet.http Package
The preceding examples have used the classes and
interfaces defined in javax.servlet, such as ServletRequest,
ServletResponse, and GenericServlet, to illustrate the
basic functionality of servlets. However, when working with HTTP, you will normally use the interfaces and classes in javax.servlet.http. As you will see, its functionality makes it easy to build servlets that work with HTTP requests and responses.

The following table summarizes the core interfaces that are provided in this package:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HttpServletRequest</td>
<td>Enables servlets to read data from an HTTP request.</td>
</tr>
<tr>
<td>HttpServletResponse</td>
<td>Enables servlets to write data to an HTTP response.</td>
</tr>
<tr>
<td>HttpSession</td>
<td>Allows session data to be read and written.</td>
</tr>
<tr>
<td>HttpSessionBindingListener</td>
<td>Informs an object that it is bound to or unbound from a session.</td>
</tr>
</tbody>
</table>

The following table summarizes the core classes that are provided in this package. The most important of these is HttpServlet. Servlet developers typically extend this class in order to process HTTP requests.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cookie</td>
<td>Allows state information to be stored on a client machine.</td>
</tr>
<tr>
<td>HttpServlet</td>
<td>Provides methods to handle HTTP requests and responses.</td>
</tr>
<tr>
<td>HttpSessionEvent</td>
<td>Encapsulates a session-changed event.</td>
</tr>
<tr>
<td>HttpSessionBindingEvent</td>
<td>Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.</td>
</tr>
</tbody>
</table>

The HttpServletRequest Interface

The HttpServletRequest interface enables a servlet to obtain information about a client request. Several of its methods are shown in Table 32-5.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getAuthType( )</td>
<td>Returns authentication scheme.</td>
</tr>
<tr>
<td>Cookie[ ] get_cookies( )</td>
<td>Returns an array of the cookies in this request.</td>
</tr>
<tr>
<td>long getDateHeader(String field)</td>
<td>Returns the value of the date header field named field.</td>
</tr>
<tr>
<td>String getHeader(String field)</td>
<td>Returns the value of the header field named field.</td>
</tr>
<tr>
<td>Enumeration&lt;String&gt; getHeaderNames( )</td>
<td>Returns an enumeration of the header names.</td>
</tr>
<tr>
<td>int getIntHeader(String field)</td>
<td>Returns the int equivalent of the header field named field.</td>
</tr>
<tr>
<td>String getMethod( )</td>
<td>Returns the HTTP method for this request.</td>
</tr>
<tr>
<td>String getPathInfo( )</td>
<td>Returns any path information that is located after the servlet path and before a query string of the URL.</td>
</tr>
<tr>
<td>String getPathTranslated( )</td>
<td>Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.</td>
</tr>
<tr>
<td>String getQueryString( )</td>
<td>Returns any query string in the URL.</td>
</tr>
<tr>
<td>String getRemoteUser( )</td>
<td>Returns the name of the user who issued this request.</td>
</tr>
<tr>
<td>String getRequestedSessionIdId( )</td>
<td>Returns the ID of the session.</td>
</tr>
<tr>
<td>String getRequestURL( )</td>
<td>Returns the URL.</td>
</tr>
<tr>
<td>StringBuffer getRequestURL( )</td>
<td>Returns the URL.</td>
</tr>
<tr>
<td>String getServletPath( )</td>
<td>Returns that part of the URL that identifies the servlet.</td>
</tr>
<tr>
<td>HttpSession getSession( )</td>
<td>Returns the session for this request. If a session does not exist, one is created and then returned.</td>
</tr>
<tr>
<td>HttpSession getSession(boolean new)</td>
<td>If new is true and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.</td>
</tr>
<tr>
<td>boolean isRequestedSessionIdFromCookie( )</td>
<td>Returns true if a cookie contains the session ID. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean isRequestedSessionIdFromURL( )</td>
<td>Returns true if the URL contains the session ID. Otherwise, returns false.</td>
</tr>
<tr>
<td>boolean isRequestedSessionIdValid( )</td>
<td>Returns true if the requested session ID is valid in the current session context.</td>
</tr>
</tbody>
</table>

Table 32-5 Various Methods Defined by HttpServletRequest

The HttpServletResponse Interface

The HttpServletResponse interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, SC_OK indicates that the HTTP request
succeeded, and SC_NOT_FOUND indicates that the requested resource is not available. Several methods of this interface are summarized in Table 32-6.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addCookie(Cookie cookie)</td>
<td>Adds cookie to the HTTP response.</td>
</tr>
<tr>
<td>boolean containsHeader(String field)</td>
<td>Returns true if the HTTP response header contains a field named field.</td>
</tr>
<tr>
<td>String encodeURL(String url)</td>
<td>Determines if the session ID must be encoded in the URL identified as url. If so, returns the modified version of url. Otherwise, returns url. All URLs generated by a servlet should be processed by this method.</td>
</tr>
<tr>
<td>String encodeRedirectURL(String url)</td>
<td>Determines if the session ID must be encoded in the URL identified as url. If so, returns the modified version of url. Otherwise, returns url. All URLs passed to sendRedirect() should be processed by this method.</td>
</tr>
<tr>
<td>void sendError(int c) throws IOException</td>
<td>Sends the error code c to the client.</td>
</tr>
<tr>
<td>void sendError(int c, String s) throws IOException</td>
<td>Sends the error code c and message s to the client.</td>
</tr>
<tr>
<td>void sendRedirect(String url) throws IOException</td>
<td>Redirects the client to url.</td>
</tr>
<tr>
<td>void setDateHeader(String field, long msec)</td>
<td>Adds field to the header with date value equal to msec (milliseconds since midnight, January 1, 1970, GMT).</td>
</tr>
<tr>
<td>void setHeader(String field, String value)</td>
<td>Adds field to the header with value equal to value.</td>
</tr>
<tr>
<td>void setIntHeader(String field, int value)</td>
<td>Adds field to the header with value equal to value.</td>
</tr>
<tr>
<td>void setStatus(int code)</td>
<td>Sets the status code for this response to code.</td>
</tr>
</tbody>
</table>

Table 32-6 Various Methods Defined by HttpServletResponse

The HttpSession Interface

The HttpSession interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in Table 32-7. All of these methods throw an IllegalStateException if the session has already been invalidated.
The HttpSessionBindingListener Interface

The HttpSessionBindingListener interface is implemented by objects that need to be notified when they are bound to or unbound from an HTTP session. The methods that are invoked when an object is bound or unbound are

```java
void valueBound(HttpSessionBindingEvent e)
void valueUnbound(HttpSessionBindingEvent e)
```

Here, `e` is the event object that describes the binding.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getattribute(String attr)</code></td>
<td>Returns the value associated with the name passed in <code>attr</code>. Returns <code>null</code> if <code>attr</code> is not found.</td>
</tr>
<tr>
<td><code>getattributeNames()</code></td>
<td>Returns an enumeration of the attribute names associated with the session.</td>
</tr>
<tr>
<td><code>getCreationTime()</code></td>
<td>Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.</td>
</tr>
<tr>
<td><code>getId()</code></td>
<td>Returns the session ID.</td>
</tr>
<tr>
<td><code>getLastAccessedTime()</code></td>
<td>Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request for this session.</td>
</tr>
<tr>
<td><code>invalidate()</code></td>
<td>Invalidates this session and removes it from the context.</td>
</tr>
<tr>
<td><code>isNew()</code></td>
<td>Returns <code>true</code> if the server created the session and it has not yet been accessed by the client.</td>
</tr>
<tr>
<td><code>removeAttribute(String attr)</code></td>
<td>Removes the attribute specified by <code>attr</code> from the session.</td>
</tr>
<tr>
<td><code>setAttribute(String attr, Object val)</code></td>
<td>Associates the value passed in <code>val</code> with the attribute name passed in <code>attr</code>.</td>
</tr>
</tbody>
</table>

Table 32-7 Various Methods Defined by HttpSession

The Cookie Class

The Cookie class encapsulates a cookie. A cookie is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user’s name, address, and other information. The user does not need to enter this data each time he or she
A servlet can write a cookie to a user’s machine via the `addCookie()` method of the `HttpServletResponse` interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user’s machine. Some of the information that is saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user’s machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the web server. Otherwise, it is not.

There is one constructor for `Cookie`. It has the signature shown here:

```
Cookie(String name, String value)
```

Here, the name and value of the cookie are supplied as arguments to the constructor. The methods of the `Cookie` class are summarized in Table 32-8.
The HttpServlet Class

The HttpServlet class extends GenericServlet. It is commonly used when developing servlets that receive and process HTTP requests. The methods defined by the HttpServlet class are summarized in Table 32-9.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object clone()</td>
<td>Returns a copy of this object.</td>
</tr>
<tr>
<td>String getComment()</td>
<td>Returns the comment.</td>
</tr>
<tr>
<td>String getDomain()</td>
<td>Returns the domain.</td>
</tr>
<tr>
<td>int getMaxAge()</td>
<td>Returns the maximum age (in seconds).</td>
</tr>
<tr>
<td>String getName()</td>
<td>Returns the name.</td>
</tr>
<tr>
<td>String getPath()</td>
<td>Returns the path.</td>
</tr>
<tr>
<td>boolean getSecure()</td>
<td>Returns true if the cookie is secure. Otherwise, returns false.</td>
</tr>
<tr>
<td>String getValue()</td>
<td>Returns the value.</td>
</tr>
<tr>
<td>int getVersion()</td>
<td>Returns the version.</td>
</tr>
<tr>
<td>boolean isHttpOnly()</td>
<td>Returns true if the cookie has the HttpOnly attribute.</td>
</tr>
<tr>
<td>void setComment(String c)</td>
<td>Sets the comment to c.</td>
</tr>
<tr>
<td>void setDomain(String d)</td>
<td>Sets the domain to d.</td>
</tr>
<tr>
<td>void setHttpOnly(boolean httpOnly)</td>
<td>If httpOnly is true, then the HttpOnly attribute is added to the cookie. If httpOnly is false, the HttpOnly attribute is removed.</td>
</tr>
<tr>
<td>void setMaxAge(int secs)</td>
<td>Sets the maximum age of the cookie to secs. This is the number of seconds after which the cookie is deleted.</td>
</tr>
<tr>
<td>void setPath(String p)</td>
<td>Sets the path to p.</td>
</tr>
<tr>
<td>void setSecure(boolean secure)</td>
<td>Sets the security flag to secure.</td>
</tr>
<tr>
<td>void setValue(String v)</td>
<td>Sets the value to v.</td>
</tr>
<tr>
<td>void setVersion(int v)</td>
<td>Sets the version to v.</td>
</tr>
</tbody>
</table>

Table 32-8 The Methods Defined by Cookie
### Table 32-9 The Methods Defined by HttpServlet

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void doDelete(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Handles an HTTP DELETE request.</td>
</tr>
<tr>
<td>void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Handles an HTTP GET request.</td>
</tr>
<tr>
<td>void doHead(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Handles an HTTP HEAD request.</td>
</tr>
<tr>
<td>void doOptions(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Handles an HTTP OPTIONS request.</td>
</tr>
<tr>
<td>void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Handles an HTTP POST request.</td>
</tr>
<tr>
<td>void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Handles an HTTP PUT request.</td>
</tr>
<tr>
<td>void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Handles an HTTP TRACE request.</td>
</tr>
<tr>
<td>long getLastModified(HttpServletRequest req)</td>
<td>Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.</td>
</tr>
<tr>
<td>void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException</td>
<td>Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.</td>
</tr>
</tbody>
</table>

---

### The HttpSessionEvent Class

HttpSessionEvent encapsulates session events. It extends EventObject and is generated when a change occurs to the session. It defines this constructor:

```
HttpSessionEvent(HttpSession session)
```

Here, session is the source of the event.

HttpSessionEvent defines one method, getSession(), which is shown here:
HttpSession getSession()

It returns the session in which the event occurred.

The HttpSessionBindingEvent Class

The HttpSessionBindingEvent class extends HttpSessionEvent. It is generated when a listener is bound or unbound in an HttpSession object. It is also generated when an attribute is bound or unbound. Here are its constructors:

HttpSessionBindingEvent(HttpSession session, String name)
HttpSessionBindingEvent(HttpSession session, String name, Object val)

Here, session is the source of the event, and name is the name associated with the object that is being bound or unbound. If an attribute is being bound or unbound, its value is passed in val.

The getName( ) method obtains the name that is being bound or unbound. It is shown here:

String getName()

The getSession( ) method, shown next, obtains the session to which the listener is being bound or unbound:

HttpSession getSession()

The getValue( ) method obtains the value of the attribute that is being bound or unbound. It is shown here:
Handling HTTP Requests and Responses

The `HttpServlet` class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are `doDelete()`, `doGet()`, `doHead()`, `doOptions()`, `doPost()`, `doPut()`, and `doTrace()`. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in `ColorGet.html`, and a servlet is defined in `ColorGetServlet.java`. The HTML source code for `ColorGet.html` is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```html
<html>
<body>
<center>
<form name="Form1"
action="http://localhost:8080/examples/servlets/servlet/ColorGetServlet">
  <B>Color:</B>
  <select name="color" size="1">
    <option value="Red">Red</option>
    <option value="Green">Green</option>
  </select>
  <input type="submit" value="Submit">
</form>
</center>
</body>
</html>
```
The source code for ColorGetServlet.java is shown in the following listing. The doGet() method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the getParameter() method of HttpServletRequest to obtain the selection that was made by the user. A response is then formulated.

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.println(color);
        pw.close();
    }
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the web.xml file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display the web page in a browser.
3. Select a color.
4. Submit the web page.
After completing these steps, the browser will display
the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request
are included as part of the URL that is sent to the web
server. Assume that the user selects the red option and
submits the form. The URL sent from the browser to the
server is


The characters to the right of the question mark are
known as the query string.

Handling HTTP POST Requests
Here we will develop a servlet that handles an HTTP
POST request. The servlet is invoked when a form on a
web page is submitted. The example contains two files.
A web page is defined in ColorPost.html, and a servlet is
defined in ColorPostServlet.java.

The HTML source code for ColorPost.html is shown in
the following listing. It is identical to ColorGet.html
except that the method parameter for the form tag
explicitly specifies that the POST method should be used,
and the action parameter for the form tag specifies a
different servlet.
The source code for ColorPostServlet.java is shown in the following listing. The doPost() method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the getParameter() method of HttpServletRequest to obtain the selection that was made by the user. A response is then formulated.

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

            String color = request.getParameter("color");
            response.setContentType("text/html");
            PrintWriter pw = response.getWriter();
            pw.println("<B>The selected color is: ");
            pw.println(color);
            pw.close();
    }
}
```

Compile the servlet and perform the same steps as described in the previous section to test it.
Parameters for an HTTP POST request are not included as part of the URL that is sent to the web server. In this example, the URL sent from the browser to the server is http://localhost:8080/examples/servlets/servlet/ColorPostServlet. The parameter names and values are sent in the body of the HTTP request.

Using Cookies

Now, let’s develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddCookie.html</td>
<td>Allows a user to specify a value for the cookie named MyCookie.</td>
</tr>
<tr>
<td>AddCookieServlet.java</td>
<td>Processes the submission of AddCookie.html.</td>
</tr>
<tr>
<td>GetCookiesServlet.java</td>
<td>Displays cookie values.</td>
</tr>
</tbody>
</table>

The HTML source code for AddCookie.html is shown in the following listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to AddCookieServlet via an HTTP POST request.

```html
<html>
<body>
<center>
<form name="Form1" method="post"
    action="http://localhost:8080/examples/servlets/servlet/AddCookieServlet">
    <B>Enter a value for MyCookie:</B>
    <input type=textbox name="data" size=25 value="">
    <input type=submit value="Submit">
</form>
</body>
</html>
```
The source code for `AddCookieServlet.java` is shown in the following listing. It gets the value of the parameter named "data". It then creates a `Cookie` object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the `addCookie()` method. A feedback message is then written to the browser.

```java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

    public void doPost(HttpServletRequest request, 
                       HttpServletRequest response) 
        throws ServletException, IOException {

        // Get parameter from HTTP request.
        String data = request.getParameter("data");

        // Create cookie.
        Cookie cookie = new Cookie("MyCookie", data);

        // Add cookie to HTTP response.
        response.addCookie(cookie);

        // Write output to browser.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>MyCookie has been set to");
        pw.println(data);
        pw.close();
    }
}
```

The source code for `GetCookiesServlet.java` is shown in the following listing. It invokes the `getCookies()` method to read any cookies that are included in the HTTP GET request. The names and values of these
cookies are then written to the HTTP response. Observe that the getName( ) and getValue( ) methods are called to obtain this information.

```java
import java.io.*;
import javax.servlet. *
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // Get cookies from header of HTTP request.
        Cookie[] cookies = request.getCookies();

        // Display these cookies.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");
        for(int i = 0; i < cookies.length; i++) {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            pw.println("name = " + name + " value = " + value);
        }
        pw.close();
    }
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the web.xml file, as previously described. Then, perform these steps to test this example:

1. Start Tomcat, if it is not already running.
2. Display AddCookie.html in a browser.
3. Enter a value for MyCookie.
4. Submit the web page.
After completing these steps, you will observe that a feedback message is displayed by the browser.

Next, request the following URL via the browser:

http://localhost:8080/examples/servlets/servlet/GetCookiesServlet

Observe that the name and value of the cookie are displayed in the browser.

In this example, an expiration date is not explicitly assigned to the cookie via the `setMaxAge()` method of `Cookie`. Therefore, the cookie expires when the browser session ends. You can experiment by using `setMaxAge()` and observe that the cookie is then saved to the disk on the client machine.

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the `getSession()` method of `HttpServletRequest`. An `HttpSession` object is returned. This object can store a set of bindings that associate names with objects. The `setAttribute()`, `getAttribute()`, `getAttributeNames()`, and `removeAttribute()` methods of `HttpSession` manage these bindings. Session state is shared by all servlets that are associated with a client.

The following servlet illustrates how to use session state. The `getSession()` method gets the current session. A new session is created if one does not already exist.
The `getAttribute()` method is called to obtain the object that is bound to the name “date”. That object is a `Date` object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A `Date` object encapsulating the current date and time is then created. The `setAttribute()` method is called to bind the name “date” to this object.

```java
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {

        // Get the HttpSession object.
        HttpSession hs = request.getSession(true);

        // Get writer.
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");

        // Display date/time of last access.
        Date date = (Date)hs.getAttribute("date");
        if(date != null) {
            pw.println("Last access: " + date + "<br>");
        }

        // Display current date/time.
        date = new Date();
        hs.setAttribute("date", date);
        pw.println("Current date: " + date);
    }
}
```

When you first request this servlet, the browser
displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.
CHAPTER 33
Financial Applets and Servlets

Despite all the large, sophisticated applications, such as word processors, databases, and accounting packages, that dominate much of the computing landscape, there has remained a class of programs that are both popular and small. These perform various financial calculations, such as the regular payments on a loan, the future value of an investment, or the remaining balance on a loan. None of these calculations are very complicated or require much code, yet they yield information that is quite useful.

As you know, Java was initially designed to support the creation of small, portable programs. Originally, these programs took the form of applets, but a few years later, servlets were added. (Recall that applets run on the local machine, inside the browser, and servlets execute on the server.) Because of their small size, many of the common financial calculations are right-sized for applets and servlets. Furthermore, including a financial applet/servlet in a web page is an amenity that many users will appreciate. A user will return again and again to a page that offers the calculation that he or she desires.

This chapter develops a number of applets that perform the financial calculations shown here:

- Regular payments on a loan
• Remaining balance on a loan
• Future value of an investment
• Initial investment needed to attain a desired future value
• Annuity from an investment
• Investment necessary for a desired annuity

The chapter ends by showing how to convert the financial applets into servlets.

Finding the Payments for a Loan
Perhaps the most popular financial calculation is the one that computes the regular payments on a loan, such as a car or house loan. The payments on a loan are found by using the following formula:

\[ \text{Payment} = \frac{(\text{intRate} \times \frac{\text{principal}}{\text{payPerYear}})}{(1 - ((\text{intRate} / \text{payPerYear}) + 1)^{-\text{payPerYear} \times \text{numYears}}) \]

where \( \text{intRate} \) specifies the interest rate, \( \text{principal} \) contains the starting balance, \( \text{payPerYear} \) specifies the number of payments per year, and \( \text{numYears} \) specifies the length of the loan in years.

The following applet called RegPay uses the preceding formula to compute the payments on a loan given the information entered by the user. Like all of the applets in this chapter, RegPay is a Swing-based applet. This means that it extends the JApplet class and uses the Swing classes to provide the user interface. Notice that it also implements the ActionListener interface.
// A simple loan calculator applet.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;

public class RegPay extends JApplet
    implements ActionListener {

    JTextField amountText, paymentText, periodText, rateText;
    JButton doit;

    double principal; // original principal
    double intRate;   // interest rate
    double numYears; // length of loan in years

    /* Number of payments per year. You could
     * allow this value to be set by the user. */
    final int payPerYear = 12;

    NumberFormat nf;
public void init() {
    try {
        SwingUtilities.invokeLater(new Runnable () {
            public void run() {
                makeGUI(); // initialize the GUI
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of "+ exc);
    }
}

// Set up and initialize the GUI.
private void makeGUI() {

    // Use a grid bag layout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
        JLabel("Compute Monthly Loan Payments");
JLabel amountLab = new JLabel("Principal ");
JLabel periodLab = new JLabel("Years ");
JLabel rateLab = new JLabel("Interest Rate ");
JLabel paymentLab = new JLabel("Monthly Payments ");

amountText = new JTextField(10);
periodText = new JTextField(10);
paymentText = new JTextField(10);
rateText = new JTextField(10);

// Payment field for display only.
paymentText.setEditable(false);

doIt = new JButton("Compute");

// Define the grid bag.
gbc.weighty = 1.0; // use a row weight of 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Anchor most components to the right.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Add all the components.
add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(paymentLab);
add(paymentText);
add(doIt);
// Register to receive action events.
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
doIt.addActionListener(this);

// Create a number format.
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
}

/* User pressed Enter on a text field or
   pressed Compute. Display the result if all
   fields are completed. */
public void actionPerformed(ActionEvent ae) {
  double result = 0.0;

  String amountStr = amountText.getText();
  String periodStr = periodText.getText();
  String rateStr = rateText.getText();

  try {
    if(amountStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0) {

The applet produced by this program is shown in Figure 33-1. To use the applet, simply enter the loan principal, the length of the loan in years, and the interest rate. The payments are assumed to be monthly. Once the information is entered, press Compute to calculate the monthly payment.

The following sections examine the code to RegPay in

```java
principal = Double.parseDouble(amountStr); numYears = Double.parseDouble(periodStr); intRate = Double.parseDouble(rateStr) / 100;
result = compute();

paymentText.setText(nf.format(result));
}
showStatus(""); // erase any previous error message
}
```

The applet produced by this program is shown in Figure 33-1. To use the applet, simply enter the loan principal, the length of the loan in years, and the interest rate. The payments are assumed to be monthly. Once the information is entered, press Compute to calculate the monthly payment.

The following sections examine the code to RegPay in
The following sections examine the code to RegPay in detail. Because all the applets in this chapter use the same basic framework, much of the explanation presented here also applies to the other applets.

![Applet Viewer: RegPay](image)

**Figure 33-1 The RegPay applet**

The RegPay Fields

RegPay begins by declaring a number of instance variables that hold references to the text fields into which the user will enter the loan information. Next, it declares the `doIt` variable that will hold a reference to the Compute button.

RegPay then declares three double variables that hold the loan values. The original principal is stored in `principal`, the interest rate is stored in `intRate`, and the length of the loan in years is stored in `numYears`. These values are entered by the user through the text fields. Next, the final integer variable `payPerYear` is declared and initialized to 12. Thus, the number of payments per
year is hard-coded to monthly because this is the way that most loans are paid. As the comments suggest, you could allow the user to enter this value, but doing so will require another text field.

The last instance variable declared by RegPay is nf, a reference to an object of type NumberFormat, which will describe the number format used for output. NumberFormat is stored in the java.text package. Although there are other ways to format numeric output, such as by using the Formatter class, NumberFormat is a good choice in this case, because the same format is used repeatedly, and this format can be set once, at the start of the program. The financial applets also offer a good opportunity to demonstrate its use.

The init( ) Method
Like all applets, the init( ) method is called when the applet first starts execution. This method simply invokes the makeGUI( ) method on the event-dispatching thread. As explained in Chapter 30, Swing-based applets must construct and interact with GUI components only through the event-dispatching thread.

The makeGUI( ) Method
The makeGUI( ) method sets up the user interface for the applet. It performs the following jobs:

1. It changes the layout manager to GridBagLayout.
2. It instantiates the various GUI components.
3. It adds the components to the grid bag.

4. It adds action listeners for the components.

Let’s now look at makeGUI( ) line by line. The method begins with these lines of code:

```java
// Use a grid bag layout.
GridBagLayout gbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);
```

This sequence creates a GridBagLayout layout manager that will be used by the applet. (For details on using GridBagLayout, see Chapter 25. ) GridBagLayout is used because it allows detailed control over the placement of controls within an applet.

Next, makeGUI( ) creates the label components, text fields, and Compute button, as shown here:

```java
JLabel heading = new
    JLabel("Compute Monthly Loan Payments");

JLabel amountLab = new JLabel("Principal ");
JLabel periodLab = new JLabel("Years ");
JLabel rateLab = new JLabel("Interest Rate ");
JLabel paymentLab = new JLabel("Monthly Payments ");

amountText = new JTextField(10);
periodText = new JTextField(10);
paymentText = new JTextField(10);
rateText = new JTextField(10);

// Payment field for display only.
paymentText.setEditable(false);

doIt = new JButton("Compute");
```

Notice that the text field that displays the monthly payment is set to read-only by calling setEditable(false).
This causes the field to be grayed and no text can be entered into the field by the user. However, the contents of the text field can still be set by calling setText(). Thus, when editing is disabled in a JTextField, the field can be used to display text, but the text cannot be changed by the user.

Next, the grid bag constraints for each component are set by the following code sequence:

```java
// Define the grid bag.
gbc.weighty = 1.0; // use a row weight of 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Anchor most components to the right.
gbc.anchor = GridBagConstraints.EAST;
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);
```
Although this seems a bit complicated at first glance, it really isn’t. Just remember that each row in the grid is specified separately. Here is how the sequence works. First, the weight of each row, contained in gbc.weighty, is set to 1. This tells the grid bag to distribute extra space evenly when there is more vertical space than needed to hold the components. Next, the gbc.gridwidth is set to REMAINDER, and gbc.anchor is set to NORTH. The label referred to by heading is added by calling setConstraints() on gbag. This sequence sets the location of heading to the top of the grid (north) and gives it the remainder of the row. Thus, after this sequence executes, the heading will be at the top of the window and on a row by itself.

Next, the four text fields and their labels are added. First, gbc.anchor is set to EAST. This causes each component to be aligned to the right. Next, gbc.gridWidth is set to RELATIVE, and the label is added. Then, gbc.gridWidth is set to REMAINDER, and the text field is added. Thus, each text field and label pair occupies one row. This process repeats until all four text field and label pairs have been added. Finally, the Compute button is added in the center.

After the grid bag constraints have been set, the components are actually added to the window by the following code:

```java
// Add all the components.
add(heading);
add(amountLab);
add(amountText);
```
Next, action listeners are registered for the three input text fields and the Compute button, as shown here:

```java
// Register to receive action events.
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
doIt.addActionListener(this);
```

Finally, a `NumberFormat` object is obtained and the format is set to two decimal digits:

```java
// Create a number format.
NF = NumberFormat.getInstance();
NF.setMinimumFractionDigits(2);
NF.setMaximumFractionDigits(2);
```

The call to the factory method `getInstance()` obtains a `NumberFormat` object suitable for the default locale. The calls to `setMinimumFractionDigits()` and `setMaximumFractionDigits()` set the minimum and maximum number of decimal digits to be displayed. Because both are set to two, this ensures that two decimal places will always be visible.

The `actionPerformed()` Method

The `actionPerformed()` method is called whenever the user presses `ENTER` when in a text field or clicks the Compute button. This method performs three main
functions: it obtains the loan information entered by the user, it calls compute() to find the loan payments, and it displays the result. Let’s now examine actionPerformed() line by line.

After declaring the result variable, actionPerformed() begins by obtaining the strings from the three user-input text fields using the following sequence:

```java
String amountStr = amountText.getText();
String periodStr = periodText.getText();
String rateStr = rateText.getText();
```

Next, it begins a try block and then verifies that all three fields actually contain information, as shown here:

```java
try {
    if(amountStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0) {
        // Recall that the user must enter the original loan amount, the number of years for the loan, and the interest rate. If all three text fields contain information, then the length of each string will be greater than zero.
        // If the user has entered all the loan data, then the numeric values corresponding to those strings are obtained and stored in the appropriate instance variable. Next, compute() is called to compute the loan payment, and the result is displayed in the read-only text field referred to by paymentText, as shown here:
        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        intRate = Double.parseDouble(rateStr) / 100;
    }
```
result = compute();
paymentText.setText(nf.format(result));

Notice the call to nf.format(result). This causes the value
i n result to be formatted as previously speci ed (with
two decimal digits) and the resulting string is returned.
This string is then used to set the text in the JTextField
specified by paymentText.
If the user has entered a nonnumeric value into one of
the text elds, then Double.parseDouble( ) will throw a
NumberFormatException. If this happens, an error
message will be displayed on the status line and the
Payment text field will be emptied, as shown here:
showStatus(""); // erase any previous error message
} catch (NumberFormatException exc) {
showStatus("Invalid Data");
paymentText.setText("");
}

Otherwise, any previously reported error is removed.
The compute( ) Method
The calculation of the loan payment takes place in
compute( ). It implements the formula shown earlier and
operates on the values in principal, intRate, numYears,
and payPerYear. It returns the result.
NOTE
The basic skeleton used by RegPay is used by all the
applets shown in this chapter.


Finding the Future Value of an Investment

Another popular financial calculation finds the future value of an investment given the initial investment, the rate of return, the number of compounding periods per year, and the number of years the investment is held. For example, you might want to know what your retirement account will be worth in 12 years if it currently contains $98,000 and has an average annual rate of return of 6 percent. The FutVal applet developed here will supply the answer.

To compute the future value, use the following formula:

$$\text{Future Value} = \text{principal} \times ((\text{rateOfRet} / \text{compPerYear}) + 1)^{\text{compPerYear} \times \text{numYears}}$$

where rateOfRet specifies the rate of return, principal contains the initial value of the investment, compPerYear specifies the number of compounding periods per year, and numYears specifies the length of the investment in years. If you use an annualized rate of return for rateOfRet, then the number of compounding periods is 1.

The following applet called FutVal uses the preceding formula to compute the future value of an investment. The applet produced by this program is shown in Figure 33-2. Aside from the computational differences within the compute( ) method, the applet is similar in operation to the RegPay applet described in the preceding section.
/ Compute the future value of an investment.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
 *
   <applet code="FutVal" width=380 height=240>
   </applet>
*/
public class FutVal extends JApplet
    implements ActionListener {

    JTextField amountText, futvalText, periodText,
        rateText, compText;
    JButton doit;

double principal; // original principal
double rateOfRet; // rate of return
double numYears; // length of investment in years
int compPerYear; // number of compoundings per year

    NumberFormat nf;

    public void init() {
        try {
            SwingUtilities.invokeLater(new Runnable () {
                public void run() {
                    makeGUI(); // initialize the GUI
                }
            });
        } catch(Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
// Set up and initialize the GUI.
private void makeGUI() {

    // Use a grid bag layout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new JLabel("Future Value of an Investment");

    JLabel amountLab = new JLabel("Principal ");
    JLabel periodLab = new JLabel("Years ");
    JLabel rateLab = new JLabel("Rate of Return ");
    JLabel futvalLab = new JLabel("Future Value of Investment");
    JLabel compLab = new JLabel("Compounding Periods per Year");

    amountText = new JTextField(10);
    periodText = new JTextField(10);
    futvalText = new JTextField(10);
    rateText = new JTextField(10);
    compText = new JTextField(10);

    // Future value field for display only.
    futvalText.setEditable(false);
    doIt = new JButton("Compute");
// Define the grid bag.
gbc.weighty = 1.0; // use a row weight of 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Anchor most components to the right.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(compLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(compText, gbc);


gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(futvalLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(futvalText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(compLab);
add(compText);
add(futvalLab);
add(futvalText);
add(doIt);

// Register to receive action events.
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
compText.addActionListener(this);
doIt.addActionListener(this);

    // Create a number format.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
}

/* User pressed Enter on a text field or pressed Compute. Display the result if all fields are completed. */
public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String amountStr = amountText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String compStr = compText.getText();

    try {
        if(amountStr.length() != 0 &&
            periodStr.length() != 0 &&
            rateStr.length() != 0 &&
            compStr.length() != 0) {

            // Implement the calculation logic here.
        }
    }
}
principal = Double.parseDouble(amountStrStr);
numYears = Double.parseDouble(periodStr);
rateOfRet = Double.parseDouble(rateStr) / 100;
compPerYear = Integer.parseInt(compStr);

result = compute();

futvalText.setText(String.format(result));
}
showStatus("'"); // erase any previous error message
}

} catch (NumberFormatException exc) {
showStatus("Invalid Data");
futvalText.setText("'");
}

// Compute the future value.
double compute() {
    double b, e;

    b = (1 + rateOfRet/compPerYear);
e = compPerYear * numYears;

    return principal * Math.pow(b, e);
}

// Applet Viewer: FutVal

Future Value of an Investment

Principal 10000
Years 5
Rate of Return 6
Compounding Periods per Year 12
Future Value of Investment 13488.50

Compute
Finding the Initial Investment Required to Achieve a Future Value

Sometimes you will want to know how large an initial investment is required to achieve some future value. For example, if you are saving for your child’s college education and you know that you will need $75,000 in five years, how much money do you need to invest at 7 percent to reach that goal? The InitInv applet developed here can answer that question.

The formula to compute an initial investment is shown here:

\[
\text{Initial Investment} = \frac{\text{targetValue}}{(((\text{rateOfRet} / \text{compPerYear}) + 1)^{\text{compPerYear} \times \text{numYears}})
\]

where rateOfRet specifies the rate of return, targetValue contains the starting balance, compPerYear specifies the number of compounding periods per year, and numYears specifies the length of the investment in years. If you use an annualized rate of return for rateOfRet, then the number of compounding periods is 1.

The following applet called InitInv uses the preceding formula to compute the initial investment required to reach a desired future value. The applet produced by this program is shown in Figure 33-3.
/* Compute the initial investment necessary for a specified future value. */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;

/*
  <applet code="InitInv" width=340 height=240>
   </applet>
*/

class InitInv extends JApplet
   implements ActionListener {
   JTextField targetText, initialText, periodText,
           rateText, compText;
   JButton doit;

double targetValue; // original targetValue
double rateOfRet;   // rate of return
double numYears;    // length of loan in years
int compPerYear;     // number of compoundings per year

NumberFormat nf;

public void init() {
   try {
      SwingUtilities.invokeLater(new Runnable () {
         public void run() {
            makeGUI(); // initialize the GUI
         }
      });
   } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
   }
}
// Set up and initialize the GUI.
private void makeGUI() {

    // Use a grid bag layout.
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbc);
    JLabel heading = new JLabel("Initial Investment Needed for " + "Future Value");

    JLabel targetLab = new JLabel("Desired Future Value ");
    JLabel periodLab = new JLabel("Years ");
    JLabel rateLab = new JLabel("Rate of Return ");
    JLabel compLab =
        new JLabel("Compounding Periods per Year ");
    JLabel initialLab =
        new JLabel("Initial Investment Required ");

    targetText = new JTextField(10);
    periodText = new JTextField(10);
    initialText = new JTextField(10);
    rateText = new JTextField(10);
    compText = new JTextField(10);

    // Initial value field for display only.
    initialText.setEditable(false);
doIt = new JButton("Compute");
// Define the grid bag.
gbc.weighty = 1.0; // use a row weight of 1

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Anchor most components to the right.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(targetLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(targetText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(compLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(compText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(initialLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(initialText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Add all the components.
add(heading);
add(targetLab);
add(targetText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(compLab);
add(compText);
add(initialLab);
add(initialText);
add(doIt);

// Register to receive action events.
targetText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
compText.addActionListener(this);
doIt.addActionListener(this);

// Create a number format.
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
/* User pressed Enter on a text field
   or pressed Compute. Display the result if all
   fields are completed. */

public void actionPerformed(ActionEvent ae)
{
    double result = 0.0;

    String targetStr = targetText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String compStr = compText.getText();

    try {
        if(targetStr.length() != 0 &&
            periodStr.length() != 0 &&
            rateStr.length() != 0 &&
            compStr.length() != 0) {

            targetValue = Double.parseDouble(targetStr);
            numYears = Double.parseDouble(periodStr);
            rateOfRet = Double.parseDouble(rateStr) / 100;
            compPerYear = Integer.parseInt(compStr);

            result = compute();

            initialText.setText(nf.format(result));
        }
        showStatus("""); // erase any previous error message
    } catch (NumberFormatException exc) {
        showStatus("Invalid Data");
        initialText.setText("");
    }
}

// Compute the required initial investment.

double compute() {
    double b, e;

    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;

    return targetValue / Math.pow(b, e);
}

Finding the Initial Investment Needed for a Desired Annuity

Another common financial calculation computes the amount of money that you must invest so that a desired annuity, in terms of a regular withdrawal, can be paid. For example, you might decide that you need $5,000 per month at retirement and that you will need that amount for 20 years. The question is how much will you need to invest to secure that annuity? The answer can be found using the following formula:

\[
\text{Initial Investment} = \left( \frac{\text{regWD} \times \text{wdPerYear}}{\text{rateOfRet}} \right) \times \left( 1 - \frac{1}{\text{rateOfRet} / \text{wdPerYear} + 1} \right) ^ {\text{wdPerYear} \times \text{numYears}}
\]
where rateOfRet specifies the rate of return, regWD contains the desired regular withdrawal, wdPerYear specifies the number of withdrawals per year, and numYears specifies the length of the annuity in years.

The Annuity applet shown here computes the initial investment required to produce the desired annuity. The applet produced by this program is shown in Figure 33-4.
public class Annuity extends JApplet
   implements ActionListener {

   JTextField regWDTText, initialText, periodText,
       rateText, numWDTText;
   JButton doIt;

double regWDAmount; // amount of each withdrawal
double rateOfRet;   // rate of return
double numYears;    // length of time in years
int numPerYear;     // number of withdrawals per year

NumberFormat nf;

public void init() {
   try {
      SwingUtilities.invokeLater(new Runnable () {
         public void run() {
            makeGUI(); // initialize the GUI
private void makeGUI() {

    // Use a grid bag layout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new JLabel("Initial Investment Needed for " + "Regular Withdrawals");

    JLabel regWDLab = new JLabel("Desired Withdrawal ");
    JLabel periodLab = new JLabel("Years ");
    JLabel rateLab = new JLabel("Rate of Return ");
    JLabel numWDLab =
        new JLabel("Number of Withdrawals per Year ");
    JLabel initialLab =
        new JLabel("Initial Investment Required ");

    regWDText = new JTextField(10);
    periodText = new JTextField(10);
    initialText = new JTextField(10);
    rateText = new JTextField(10);
    numWDText = new JTextField(10);
// Initial investment field for display only.
initialText.setEditable(false);

doIt = new JButton("Compute");

// Define the grid bag.
gbc.weighty = 1.0; // use a row weight of 1

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Anchor most components to the right.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(regWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(regWDTxt, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(numWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(numWDTText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(initialLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(initialText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Add all the components.
add(heading);
add(regWDLab);
add(regWDTText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(numWDLab);
add(numWDTText);
add(initialLab);
add(initialText);
add(doIt);

// Register to receive text field action events.
regWDTText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
umWDTText.addActionListener(this);
doIt.addActionListener(this);
// Create a number format.
NumberFormat nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
}

/* User pressed Enter on a text field or pressed Compute. Display the result if all fields are completed. */
public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String regWDStr = regWDText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();

    try {
        if(regWDStr.length() != 0 &&
            periodStr.length() != 0 &&
            rateStr.length() != 0 &&
            numWDStr.length() != 0) {

regWDAmount = Double.parseDouble(regWDStr);
numYears = Double.parseDouble(periodStr);
rateOfRet = Double.parseDouble(rateStr) / 100;
numPerYear = Integer.parseInt(numWDStr);

result = compute();

initialText.setText(nf.format(result));
}
showStatus("""); // erase any previous error message
} catch (NumberFormatException exc) {
  showStatus("Invalid Data");
  initialText.setText(""");
}

// Compute the required initial investment.
double compute() {
  double b, e;
  double t1, t2;

  t1 = (regWDAmount * numPerYear) / rateOfRet;

  b = (1 + rateOfRet/numPerYear);
  e = numPerYear * numYears;

  t2 = 1 - (1 / Math.pow(b, e));

  return t1 * t2;
}
Finding the Maximum Annuity for a Given Investment

Another annuity calculation computes the maximum annuity (in terms of a regular withdrawal) available from a given investment over a specified period of time. For example, if you have $500,000 in a retirement account, how much can you take out each month for 20 years, assuming a 6 percent rate of return? The formula that computes the maximum withdrawal is shown here:

$$\text{Maximum Withdrawal} = \text{principal} \times \left( \frac{((\text{rateOfRet} / \text{wdPerYear})}{-1 + ((\text{rateOfRet} / \text{wdPerYear}) + 1)^{\text{wdPerYear} \times \text{numYears}}) + \frac{(\text{rateOfRet} / \text{wdPerYear})}{(rateOfRet / wdPerYear)} \right)$$

where rateOfRet specifies the rate of return, principal contains the value of the initial investment, wdPerYear specifies the number of withdrawals per year, and numYears specifies the length of the annuity in years.

The MaxWD applet shown next computes the maximum periodic withdrawals that can be made over a
specified length of time for an assumed rate of return. The applet produced by this program is shown in Figure 33-5.

```java
/* Compute the maximum annuity that can be withdrawn from an investment over a period of time. */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;

/*
   <applet code="MaxWD" width=340 height=260>
</applet>
*/

public class MaxWD extends JApplet
    implements ActionListener {

    JTextField maxWDTText, orgPText, periodText,
                rateText, numWDTText;
    JButton doit;

double principal; // initial principal
double rateOfRet; // annual rate of return
double numYears; // length of time in years
int numPerYear; // number of withdrawals per year

    NumberFormat nf;

public void init() {
    try {
        SwingUtilities.invokeLater(new Runnable () {
            public void run() {
                makeGUI(); // initialize the GUI
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of " + exc);
    }
}
```
// Set up and initialize the GUI.
private void makeGUI() {

    // Use a grid bag layout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
        JLabel("Maximum Regular Withdrawals");

    JLabel orgPLab = new JLabel("Original Principal ");
    JLabel periodLab = new JLabel("Years ");
    JLabel rateLab = new JLabel("Rate of Return ");
    JLabel numWDLab =
        new JLabel("Number of Withdrawals per Year ");
    JLabel maxWDLab = new JLabel("Maximum Withdrawal ");

    maxWDDText = new JTextField(10);
    periodText = new JTextField(10);
    orgPTText = new JTextField(10);
    rateText = new JTextField(10);
    numWDDText = new JTextField(10);

    // Max withdrawal field for display only.
    maxWDDText.setEditable(false);

    doIt = new JButton("Compute");

    // Define the grid bag.
    gbc.weighty = 1.0; // use a row weight of 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Anchor most components to the right.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(orgPLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(orgPText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(numWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(numWDText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(maxWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(maxWDText, gbc);
gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Add all the components.
add(heading);
add(orgPLabel);
add(orgPText);
add(periodLabel);
add(periodText);
add(rateLabel);
add(rateText);
add(numWDLabel);
add(numWDText);
add(maxWDLabel);
add(maxWDText);
add(doIt);

// Register to receive action events.
orgPText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
umWDText.addActionListener(this);
doIt.addActionListener(this);

// Create a number format.
fraction = NumberFormat.getInstance();
fraction.setMinimumFractionDigits(2);
fraction.setMaximumFractionDigits(2);
public void actionPerformed(ActionEvent ae) {
    double result = 0.0;
    String orgPStr = orgPText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();

    try {
        if (orgPStr.length() != 0 &&
            periodStr.length() != 0 &&
            rateStr.length() != 0 &&
            numWDStr.length() != 0) {

            principal = Double.parseDouble(orgPStr);
            numYears = Double.parseDouble(periodStr);
            rateOfRet = Double.parseDouble(rateStr) / 100;
            numPerYear = Integer.parseInt(numWDStr);

            result = compute();

            maxWDText.setText(nf.format(result));
        }
    }
}
Finding the Remaining Balance on a Loan

```java
showStatus(" "); // erase any previous error message
} catch (NumberFormatException exc) {
  showStatus("Invalid Data");
  maxWDText.setText(" ");
}

// Compute the maximum regular withdrawals.
double compute() {
  double b, e;
  double t1, t2;

  t1 = rateOfReturn / numPerYear;
  b = (1 + t1);
  e = numPerYear * numYears;
  t2 = Math.pow(b, e) - 1;
  return principal * (t1/t2 + t1);
}

Figure 33-5 The MaxWD applet

Finding the Remaining Balance on a Loan
Often, you will want to know the remaining balance on a loan. This is easily calculated if you know the original principal, the interest rate, and the number of payments made. To find the remaining balance, you must sum the payments, subtracting from each payment the amount allocated to interest, and then subtract that result from the principal.

The RemBal applet, shown next, finds the remaining balance of a loan. The applet produced by this program is shown in Figure 33-6.

```java
// Find the remaining balance on a loan.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
   <applet code="RemBal" width=340 height=260>
     </applet>
*/

public class RemBal extends JApplet
    implements ActionListener {

    JTextField orgPText, paymentText, remBalText, 
    rateText, numPayText;
    JButton doIt;

    double orgPrincipal; // original principal
    double intRate;      // interest rate
    double payment;      // amount of each payment
    double numPayments;  // number of payments made

    /* Number of payments per year. You could allow this value to be set by the user. */
    final int payPerYear = 12;
```
NumberFormat nf;

public void init() {
    try {
        SwingUtilities.invokeLater(new Runnable () {
            public void run() {
                makeGUI(); // initialize the GUI
            }
        });
    } catch(Exception exc) {
        System.out.println("Can't create because of "+ exc);
    }
}

// Set up and initialize the GUI
private void makeGUI() {

    // Use a grid bag layout.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
        JLabel("Find Loan Balance ");

    JLabel orgPLab = new JLabel("Original Principal ");
    JLabel paymentLab = new JLabel("Amount of Payment ");
    JLabel numPayLab = new JLabel("Number of Payments Made ");
    JLabel rateLab = new JLabel("Interest Rate ");
    JLabel remBalLab = new JLabel("Remaining Balance ");
orgPText = new JTextField(10);
paymentText = new JTextField(10);
remBalText = new JTextField(10);
rateText = new JTextField(10);
numPayText = new JTextField(10);

// Payment field for display only.
remBalText.setEditable(false);

doIt = new JButton("Compute");

// Define the grid bag.
gbc.weighty = 1.0; // use a row weight of 1

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Anchor most components to the right.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(orgPLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(orgPText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);
gbc.gridwidth = GridBagConstraints.RELATIVE;
gbc.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbc.setConstraints(numPayLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.setConstraints(numPayText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbc.setConstraints(remBalLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.setConstraints(remBalText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbc.setConstraints(doIt, gbc);

// Add all the components.
add(heading);
add(orgPLab);
add(orgPText);
add(paymentLab);
add(paymentText);
add(numPayLab);
add(numPayText);
add(rateLab);
add(rateText);
add(rateText);
add(remBalLab);
add(remBalText);
add(doIt);
// Register to receive action events.
orgPText.addActionListener(this);
numPayText.addActionListener(this);
rateText.addActionListener(this);
paymentText.addActionListener(this);
doIt.addActionListener(this);

// Create a number format.
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
}

/* User pressed Enter on a text field or pressed Compute. Display the result if all fields are completed. */
public void actionPerformed(ActionEvent ae) {
  double result = 0.0;

  String orgPStr = orgPText.getText();
  String numPayStr = numPayText.getText();
  String rateStr = rateText.getText();
  String payStr = paymentText.getText();

  try {
    if(orgPStr.length() != 0 &&
        numPayStr.length() != 0 &&
        rateStr.length() != 0 &&
        payStr.length() != 0) {

orgPrincipal = Double.parseDouble(orgPStr);
numPayments = Double.parseDouble(numPayStr);
intRate = Double.parseDouble(rateStrStr) / 100;
payment = Double.parseDouble(payStrStr);

result = compute();

remBalText.setText(nf.format(result));
}
showStatus("""); // erase any previous error message
} catch (NumberFormatException exc) {
  showStatus("Invalid Data");
  remBalText.setText("");
}

// Compute the loan balance.
double compute() {
  double bal = orgPrincipal;
  double rate = intRate / payPerYear;

  for(int i = 0; i < numPayments; i++)
    bal -= payment - (bal * rate);

  return bal;
}
Creating Financial Servlets

Although applets are easy to create and use, they are only one half of the Java Internet equation. The other half is servlets. Servlets execute on the server side of the connection, and they are more appropriate for some applications. Because many readers may want to use servlets rather than applets in their commercial applications, the remainder of this chapter shows how to convert the financial applets into servlets.

Because all the financial applets use the same basic skeleton, we will walk through the conversion of only one applet: RegPay. You can then apply the same basic process to convert any of the other applets into servlets on your own. As you will see, it’s not hard to do.

NOTE
For information on creating, testing, and running servlets, see Chapter 32.

Converting the RegPay Applet into a Servlet

It is fairly easy to convert the RegPay loan calculating applet into a servlet. First, the servlet must import the javax.servlet and javax.servlet.http packages. It must also extend HttpServlet, not JApplet. Next, you must remove all the GUI code. Then, you must add the code that obtains the parameters passed to the servlet by the HTML that calls the servlet. Finally, the servlet must send
the HTML that displays the results. The basic financial calculations remain the same. It is only the way data is obtained and displayed that changes.

The RegPayS Servlet

The following RegPayS class is the servlet version of the RegPay applet. As the code is written, it assumes that RegPayS.class will be stored in Tomcat’s example servlets directory, as described in Chapter 32. Remember to enter its name into the web.xml file, also as described in Chapter 32.
// A simple loan calculator servlet.
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.*;

public class RegPayS extends HttpServlet {
    double principal; // original principal
    double intRate;    // interest rate
    double numYears;   // length of loan in years

    /* Number of payments per year. You could allow this value to be set by the user. */
    final int payPerYear = 12;

    NumberFormat nf;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String payStr = "";

        // Create a number format.
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(2);
        nf.setMaximumFractionDigits(2);
// Get the parameters.
String amountStr = request.getParameter("amount");
String periodStr = request.getParameter("period");
String rateStr = request.getParameter("rate");

try {
    if(amountStr != null && periodStr != null && 
        rateStr != null) {
        principal = Double.parseDouble(amountStr);

        numYears = Double.parseDouble(periodStr);
        intRate = Double.parseDouble(rateStr) / 100;

        payStr = nf.format(compute());
    } else { // one or more parameters missing
        amountStr = "";
        periodStr = "";
        rateStr = "";
    }
} catch (NumberFormatException exc) {
    // Take appropriate action here.
}

// Set the content type.
response.setContentType("text/html");
Get the output stream.
PrintWriter pw = response.getWriter();

Display the necessary HTML.
pw.print("<html><body> <left>" +
    " <form name="Form1" >
    " action="http://localhost:8080/" +
    " examples/servlets/servlet/RegPayS" +
    " <B>Enter amount to finance:<B> " +
    " <input type=\"textbox\" name="\"amount\" +
    " size=12 value="\">\n    ");
pw.print(amountStr + "\\>");
pw.print("<BR><B>Enter term in years:<B> " +
    " <input type=\"textbox\" name="\"period\" +
    " size=12 value="\">\n    ");
pw.println(periodStr + "\\>");
pw.print("<BR><B>Enter interest rate:<B> " +
    " <input type=\"textbox\" name="\"rate\" +
    " size=12 value="\">\n    ");
pw.print(rateStr + "\\>");
pw.print("<BR><B>Monthly Payment:<B> " +
    " <input READONLY type=\"textbox\" +
    " name="\"payment\" size=12 value="\">\n    ");
pw.print(payStr + "\\>");
pw.print("<BR><P><input type=submit value="\"Submit\"">\n    ");
pw.println("</form> </body> </html>";

Compute the loan payment.
double compute() {
    double numer;
    double denom;
    double b, e;
The first thing to notice about RegPayS is that it has only two methods: `doGet()` and `compute()`. The `compute()` method is the same as that used by the applet. The `doGet()` method is defined by the `HttpServlet` class, which RegPayS extends. This method is called by the server when the servlet must respond to a GET request. Notice that it is passed a reference to the `HttpServletRequest` and `HttpServletResponse` objects associated with the request.

From the request parameter, the servlet obtains the arguments associated with the request. It does this by calling `getParameter()`. The parameter is returned in its string form. Thus, a numeric value must be manually converted into its binary format. If no parameter is available, a null is returned.

From the response object, the servlet obtains a stream to which response information can be written. The response is then returned to the browser by outputting to that stream. Prior to obtaining a `PrintWriter` to the response stream, the output type should be set to `text/html` by calling `setContentType()`.
RegPayS can be called with or without parameters. If called without parameters, the servlet responds with the necessary HTML to display an empty loan calculator form. Otherwise, if called with all needed parameters, RegPayS calculates the loan payment and redisplays the form, with the payment field filled in.

The simplest way to invoke RegPayS is to link to its URL without passing any parameters. For example, assuming that you are using Tomcat, you can use this line to execute it:

```html
<A HREF = "http://localhost:8080/examples/servlets/servlet/RegPayS">Loan Calculator</A>
```

This displays a link called Loan Calculator that links to the RegPayS servlet in the Tomcat example servlets directory. Notice that no parameters are passed. This causes RegPayS to return the complete HTML that displays an empty loan calculator page.

You can also invoke RegPayS by first displaying an empty form manually, if you like. This approach is shown here, again using Tomcat’s example servlets directory:

```html
<html>
<body>
<form name="Form1"
    action="http://localhost:8080/examples/servlets/servlet/RegPayS">
    <B>Enter amount to finance:</B>
    <input type=textbox name="amount" size=12 value="">
    <BR>
    <B>Enter term in years:</B>
    <input type=textbox name="period" size=12 value="">
    <BR>
</form>
</body>
</html>
```
Some Things to Try
The first thing you might want to try is converting the other financial applets into servlets. Because all the financial applets are built on the same skeleton, simply follow the same approach as used by RegPayS. There are many other financial calculations that you might find useful to implement as applets or servlets, such as the rate of return of an investment or the amount of a regular deposit needed over time to reach a future value. You could also print a loan amortization chart. You might want to try creating a larger application that offers all the calculations presented in this chapter, allowing the user to select the desired calculation from a menu.
Creating a Download Manager in Java

Have you ever had an Internet download interrupted, putting you back at square one? If you have ever connected to the Internet with a dialup connection, it's very likely that you've run into this all too common nuisance. Everything from call-waiting disconnects to computer crashes can leave a download dead in its tracks. However, even with a high-speed connection, transmission disruptions can still occur. To say the least, restarting a download from scratch over and over can be a very time-consuming and frustrating experience.

A sometimes overlooked fact is that many interrupted downloads can be resumed. This allows you to recommence downloading from the point at which a download terminates instead of having to begin anew. In this chapter, a tool called Download Manager is developed that manages Internet downloads for you and makes simple work of resuming interrupted downloads. It also lets you pause and then resume a download, and manage multiple downloads, simultaneously.

At the core of the Download Manager's usefulness is its ability to take advantage of downloading only specific portions of a file. In a classic download scenario, a whole file is downloaded from beginning to end. If the transmission of the file is interrupted for any reason, the progress made toward completing the downloading of the file is lost. The Download Manager, however, can pick up from where an interruption occurs and then download only the file’s remaining fragment. Not all downloads are created equal, though, and there are some that simply cannot be restarted. Details on which files are and aren’t resumable are explained in the following section.

Not only is the Download Manager a useful utility, it
is an excellent illustration of the power and succinctness of Java’s built-in APIs—especially as they apply to interfacing to the Internet. Because the Internet was a driving force behind the creation of Java, it should come as no surprise that Java’s networking capabilities are unsurpassed. For example, attempting to create the Download Manager in another language, such as C++, would entail significantly more trouble and effort.

Understanding Internet Downloads
To understand and appreciate the Download Manager, it’s necessary to shed some light on how Internet downloads really work.

Internet downloads in their simplest form are merely client/server transactions. The client, your browser, requests to download a file from a server on the Internet. The server then responds by sending the requested file to your browser. In order for clients to communicate with servers, they must have an established protocol for doing so. The most common protocols for downloading files are File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP). FTP is usually associated generically with exchanging files between computers, whereas HTTP is usually associated specifically with transferring web pages and their related files (that is, graphics, sounds, and so on). Over time, as the World Wide Web has grown in popularity, HTTP has become the dominant protocol for downloading files from the Internet. FTP is definitely not extinct, though.

For brevity’s sake, the Download Manager developed in this chapter will only support HTTP downloads. Nonetheless, adding support for FTP would be an excellent exercise for extending the code. HTTP downloads come in two forms: resumable (HTTP 1.1) and nonresumable (HTTP 1.0). The difference between these two forms lies in the way files can be requested
from servers. With the antiquated HTTP 1.0, a client can only request that a server send it a file, whereas with HTTP 1.1, a client can request that a server send it a complete file or only a specific portion of a file. This is the feature the Download Manager is built on.

An Overview of the Download Manager
The Download Manager uses a simple yet effective GUI interface built with Java’s Swing libraries. The Download Manager window is shown in Figure 34-1. The use of Swing gives the interface a crisp, modern look and feel.

   The GUI maintains a list of downloads that are currently being managed. Each download in the list reports its URL, size of the file in bytes, progress as a percentage toward completion, and current status. The downloads can each be in one of the following different states: Downloading, Paused, Complete, Error, or Cancelled. The GUI also has controls for adding downloads to the list and for changing the state of each download in the list. When a download in the list is selected, depending on its current state, it can be paused, resumed, cancelled, or removed from the list altogether.

   The Download Manager is broken into a few classes for natural separation of functional components. These are the Download, DownloadsTableModel, ProgressRenderer, and DownloadManager classes, respectively. The DownloadManager class is responsible for the GUI interface and makes use of the DownloadsTableModel and ProgressRenderer classes for displaying the current list of downloads. The Download class represents a “managed” download and is responsible for performing the actual downloading of a file. In the following sections, we’ll walk through each of these classes in detail, highlighting their inner workings and explaining how they relate to each other.
The Download Class

The Download class is the workhorse of the Download Manager. Its primary purpose is to download a file and save that file’s contents to disk. Each time a new download is added to the Download Manager, a new Download object is instantiated to handle the download.

The Download Manager has the ability to download multiple files at once. To achieve this, it’s necessary for each of the simultaneous downloads to run independently. It’s also necessary for each individual download to manage its own state so that it can be reflected in the GUI. This is accomplished with the Download class.

The entire code for Download is shown here. Notice that it extends Observable and implements Runnable. Each part is examined in detail in the sections that follow.
// This class downloads a file from a URL.
class Download extends Observable implements Runnable {
    // Max size of download buffer.
    private static final int MAX_BUFFER_SIZE = 1024;

    // These are the status names.
    public static final String STATUSES[] = {"Downloading", "Paused", "Complete", "Cancelled", "Error"};

    // These are the status codes.
    public static final int DOWNLOADING = 0;
    public static final int PAUSED = 1;
    public static final int COMPLETE = 2;
    public static final int CANCELLED = 3;
    public static final int ERROR = 4;
    private URL url; // download URL
    private int size; // size of download in bytes
    private int downloaded; // number of bytes downloaded
    private int status; // current status of download

    // Constructor for Download.
    public Download(URL url) {
        this.url = url;
        size = -1;
        downloaded = 0;
        status = DOWNLOADING;

        // Begin the download.
        download();
    }

    // Get this download's URL.
    public String getUrl() {
        return url.toString();
    }
}
public int getSize() {
    return size;
}

public float getProgress() {
    return ((float) downloaded / size) * 100;
}

public int getStatus() {
    return status;
}

public void pause() {
    status = PAUSED;
    stateChanged();
}

public void resume() {
    status = DOWNLOADING;
    stateChanged();
    download();
}

public void cancel() {
    status = CANCELLED;
    stateChanged();
}
private void error() {
    status = ERROR;
    stateChanged();
}

private void download() {
    Thread thread = new Thread(this);
    thread.start();
}

private String getFileName(URL url) {
    String fileName = url.getFile();
    return fileName.substring(fileName.lastIndexOf('/') + 1);
}

public void run() {
    RandomAccessFile file = null;
    InputStream stream = null;
    try {
        // Open connection to URL.
        HttpURLConnection connection =
            (HttpURLConnection) url.openConnection();

        // Specify what portion of file to download.
        connection.setRequestProperty("Range", "bytes=" + downloaded + ":-1");

        // Connect to server.
        connection.connect();

        // Make sure response code is in the 200 range.
        if (connection.getResponseCode() / 100 != 2) {
            error();
        }
    }
}
// Check for valid content length.
int contentLength = connection.getContentLength();
if (contentLength < 1) {
    error();
}

/* Set the size for this download if it
hasn't been already set. */
if (size == -1) {
    size = contentLength;
    stateChanged();
}

// Open file and seek to the end of it.
file = new RandomAccessFile(getFileName(url), "rw");
file.seek(downloaded);

stream = connection.getInputStream();
while (status == DOWNLOADING) {
    /* Size buffer according to how much of the
    file is left to download. */
    byte buffer[];
    if (size - downloaded > MAX_BUFFER_SIZE) {
        buffer = new byte[MAX_BUFFER_SIZE];
    } else {
        buffer = new byte[size - downloaded];
    }

    // Read from server into buffer.
    int read = stream.read(buffer);
    if (read == -1)
        break;

    // Write buffer to file.
    file.write(buffer, 0, read);
    downloaded += read;
    stateChanged();
}
The Download Variables

Download begins by declaring several static final variables that specify the various constants used by the class. Next, four instance variables are declared. The url variable holds the Internet URL for the file being downloaded; the size variable holds the size of the download file in bytes; the downloaded variable holds the number of bytes that have been downloaded thus far; and the status variable indicates the download's current status.

The Download Constructor

Download's constructor is passed a reference to the URL to download in the form of a URL object, which is assigned to the url instance variable. It then sets the remaining instance variables to their initial states and
calls the download( ) method. Notice that size is set to −1 to indicate there is no size yet.

**The download( ) Method**

The download( ) method creates a new Thread object, passing it a reference to the invoking Download instance. As mentioned before, it’s necessary for each download to run independently. In order for the Download class to act alone, it must execute in its own thread. Java has excellent built-in support for threads and makes using them a snap. To use threads, the Download class simply implements the Runnable interface by overriding the run( ) method. After the download( ) method has instantiated a new Thread instance, passing its constructor the Runnable Download class, it calls the thread’s start( ) method. Invoking the start( ) method causes the Runnable instance’s (the Download class’) run( ) method to be executed.

**The run( ) Method**

When the run( ) method executes, the actual downloading gets under way. Because of its size and importance, we will examine it closely, line by line. The run( ) method begins with these lines:

```java
RandomAccessFile file = null;
InputStream stream = null;
try {
    // Open connection to URL.
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
```

First, run( ) sets up variables for the network stream that the download’s contents will be read from and sets up the file that the download’s contents will be written to. Next, a connection to the download’s URL is opened by calling url.openConnection( ). Since we know that the Download Manager supports only HTTP downloads, the connection is cast to the HttpURLConnection type. Casting the connection as an HttpURLConnection allows
us to take advantage of HTTP-specific connection features such as the getResponseCode() method. Note that calling url.openConnection() does not actually create a connection to the URL's server. It simply creates a new URLConnection instance associated with the URL that later will be used to connect to the server.

After the HttpURLConnection has been created, the connection request property is set by calling connection.setRequestProperty(), as shown here:

```java
// Specify what portion of file to download.
connection.setRequestProperty("Range",
  "bytes=" + downloaded + "-");
```

Setting request properties allows extra request information to be sent to the server the download will be coming from. In this case, the “Range” property is set. This is critically important, as the “Range” property specifies the range of bytes that is being requested for download from the server. Normally, all of a file’s bytes are downloaded at once. However, if a download has been interrupted or paused, only the download’s remaining bytes should be retrieved. Setting the “Range” property is the foundation for the Download Manager’s operation.

The “Range” property is specified in this form:

```
start-byte – end-byte
```

For example, “0 – 12345”. However, the end byte of the range is optional. If the end byte is absent, the range ends at the end of the file. The run() method never specifies the end byte because downloads must run until the entire range is downloaded, unless paused or interrupted.

The next few lines are shown here:

```java
// Connect to server.
connection.connect();
```
// Make sure response code is in the 200 range.
if (connection.getResponseCode() / 100 != 2) {
    error();
}

// Check for valid content length.
int contentLength = connection.getContentLength();
if (contentLength < 1) {
    error();
}

The connection.connect() method is called to make the actual connection to the download’s server. Next, the response code returned by the server is checked. The HTTP protocol has a list of response codes that indicate a server’s response to a request. HTTP response codes are organized into numeric ranges of 100, and the 200 range indicates success. The server’s response code is validated for being in the 200 range by calling connection.getResponseCode() and dividing by 100. If the value of this division is 2, then the connection was successful.

Next, run() gets the content length by calling connection.getContentLength(). The content length represents the number of bytes in the requested file. If the content length is less than 1, the error() method is called. The error() method updates the download’s status to ERROR, and then calls stateChanged(). The stateChanged() method will be described in detail later.

After getting the content length, the following code checks to see if it has already been assigned to the size variable:

    /* Set the size for this download if it hasn't been already set. */
    if (size == -1) {
        size = contentLength;
        stateChanged();
    }

As you can see, instead of assigning the content length to the size variable unconditionally, it only gets assigned if it hasn’t already been given a value. The reason for this
is because the content length reflects how many bytes the server will be sending. If anything other than a 0-based start range is specified, the content length will represent only a portion of the file's size. The size variable has to be set to the complete size of the download’s file.

The next few lines of code shown here create a new RandomAccessFile using the filename portion of the download’s URL that is retrieved with a call to the getFileName() method:

```
// Open file and seek to the end of it.
file = new RandomAccessFile(getFileName(url), "rw");
file.seek(downloaded);
```

The RandomAccessFile is opened in “rw” mode, which specifies that the file can be written to and read from. Once the file is open, run() seeks to the end of the file by calling the file.seek() method, passing in the downloaded variable. This tells the file to position itself at the number of bytes that have been downloaded—in other words, at the end. It’s necessary to position the file at the end in case a download has been resumed. If a download is resumed, the newly downloaded bytes are appended to the file and they don’t overwrite any previously downloaded bytes. After preparing the output file, a network stream handle to the open server connection is obtained by calling connection.getInputStream(), as shown here:

```
stream = connection.getInputStream();
```

The heart of all the action begins next with a while loop:

```
while (status == DOWNLOADING) {
    /* Size buffer according to how much of the file is left to download. */
    byte buffer[];
    if (size - downloaded > MAX_BUFFER_SIZE) {
        buffer = new byte[MAX_BUFFER_SIZE];
    } else {
        buffer = new byte[size - downloaded];
    }
```
This loop is set up to run until the download's status variable changes from DOWNLOADING. Inside the loop, a byte buffer array is created to hold the bytes that will be downloaded. The buffer is sized according to how much of the download is left to complete. If there is more left to download than the MAX_BUFFER_SIZE, the MAX_BUFFER_SIZE is used to size the buffer. Otherwise, the buffer is sized exactly at the number of bytes left to download. Once the buffer is sized appropriately, the downloading takes place with a stream.read( ) call. This call reads bytes from the server and places them into the buffer, returning the count of how many bytes were actually read. If the number of bytes read equals −1, then downloading has completed and the loop is exited. Otherwise, downloading is not finished and the bytes that have been read are written to disk with a call to file.write( ). Then the downloaded variable is updated, reflecting the number of bytes downloaded thus far. Finally, inside the loop, the stateChanged( ) method is invoked. More on this later.

After the loop has exited, the following code checks to see why the loop was exited:

```java
/* Change status to complete if this point was reached because downloading has finished. */
if (status == DOWNLOADING) {
    status = COMPLETE;
    stateChanged();
}
```

If the download's status is still DOWNLOADING, this means that the loop exited because downloading has
been completed. Otherwise, the loop was exited because the download’s status changed to something other than DOWNLOADING.

The run( ) method wraps up with the catch and finally blocks shown here:

```java
} catch (Exception e) {
    error();
} finally {
    // Close file.
    if (file != null) {
        try {
            file.close();
        } catch (Exception e) {} // Catch and ignore exceptions related to closing the file.
    }
    // Close connection to server.
    if (stream != null) {
        try {
            stream.close();
        } catch (Exception e) {} // Catch and ignore exceptions related to closing the stream.
    }
}
```

If an exception is thrown during the download process, the catch block captures the exception and calls the error( ) method. The finally block ensures that if the file and stream connections have been opened, they get closed whether an exception has been thrown or not. As an exercise, you might try changing this code to use the new try-with-resources statement to manage these resources.

The stateChanged( ) Method

In order for the Download Manager to display up-to-date information on each of the downloads it's managing, it has to know each time a download’s information changes. To handle this, the Observer software design pattern is used. The Observer pattern is analogous to an announcement’s mailing list where several people register to receive announcements. Each time there’s a new announcement, each person on the list receives a message with the announcement. In the Observer pattern’s case, there’s an observed class with which
observer classes can register themselves to receive change notifications.

The Download class employs the Observer pattern by extending Java’s built-in Observable utility class. Extending the Observable class allows classes that implement Java’s Observer interface to register themselves with the Download class to receive change notifications. Each time the Download class needs to notify its registered Observers of a change, the `stateChanged()` method is invoked. The `stateChanged()` method first calls the Observable class’ `setChanged()` method to flag the class as having been changed. Next, the `stateChanged()` method calls Observable’s `notifyObservers()` method, which broadcasts the change notification to the registered Observers.

Action and Accessor Methods
The Download class has numerous action and accessor methods for controlling a download and getting data from it. Each of the `pause()`, `resume()`, and `cancel()` action methods simply does as its name implies: pauses, resumes, or cancels the download, respectively. Similarly, the `error()` method marks the download as having an error. The `getUrl()`, `getSize()`, `getProgress()`, and `getStatus()` accessor methods each return their current respective values.

The ProgressRenderer Class
The ProgressRenderer class is a small utility class that is used to render the current progress of a download listed in the GUI’s “Downloads” JTable instance. Normally, a JTable instance renders each cell’s data as text. However, often it’s particularly useful to render a cell’s data as something other than text. In the Download Manager’s case, we want to render each of the table’s Progress column cells as progress bars. The ProgressRenderer class shown here makes that possible. Notice that it extends
JProgressBar and implements TableCellRenderer:

```java
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;

// This class renders a JProgressBar in a table cell.
class ProgressRenderer extends JProgressBar
    implements TableCellRenderer
{
    // Constructor for ProgressRenderer.
    public ProgressRenderer(int min, int max) {
        super(min, max);
    }

    /* Returns this JProgressBar as the renderer
     for the given table cell. */
    public Component getTableCellRendererComponent(
        JTable table, Object value, boolean isSelected,
        boolean hasFocus, int row, int column)
    {
        // Set JProgressBar's percent complete value.
        setValue(((int) ((Float) value).floatValue()));
        return this;
    }
}
```

The ProgressRenderer class takes advantage of the fact that Swing’s JTable class has a rendering system that can accept “plug-ins” for rendering table cells. To plug into this rendering system, first, the ProgressRenderer class has to implement Swing’s TableCellRenderer interface. Second, a ProgressRenderer instance has to be registered with a JTable instance; doing so instructs the JTable instance as to which cells should be rendered with the “plug-in.”

Implementing the TableCellRenderer interface requires the class to override the `getTableCellRendererComponent( )` method. The `getTableCellRendererComponent( )` method is invoked each time a JTable instance goes to render a cell for which this class has been registered. This method is passed several variables, but in this case, only the value variable is used. The value variable holds the data for the cell being rendered and is passed to JProgressBar’s `setValue( )` method. The
getTableCellRendererComponent() method wraps up by returning a reference to its class. This works because the ProgressRenderer class is a subclass of JProgressBar, which is a descendent of the AWT Component class.

The DownloadsTableModel Class

The DownloadsTableModel class houses the Download Manager’s list of downloads and is the backing data source for the GUI’s “Downloads” JTable instance.

The DownloadsTableModel class is shown here. Notice that it extends AbstractTableModel and implements the Observer interface:

```java
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

// This class manages the download table's data.
class DownloadsTableModel extends AbstractTableModel
    implements Observer
{

    // These are the names for the table's columns.
    private static final String[] columnNames = {"URL", "Size", "Progress", "Status"};

    // These are the classes for each column's values.
    private static final Class[] columnClasses = {String.class, String.class, JProgressBar.class, String.class};

    // The table's list of downloads.
    private ArrayList<Download> downloadList = new ArrayList<Download>();

    // Add a new download to the table.
    public void addDownload(Download download) {
        // Register to be notified when the download changes.
        download.addObserver(this);
        downloadList.add(download);

        // Fire table row insertion notification to table.
        fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
    }
```
The DownloadsTableModel class essentially is a utility
class utilized by the “Downloads” JTable instance for managing data in the table. When the JTable instance is initialized, it is passed a DownloadsTableModel instance. The JTable then proceeds to call several methods on the DownloadsTableModel instance to populate itself. The getColumnCount( ) method is called to retrieve the number of columns in the table. Similarly, getRowCount( ) is used to retrieve the number of rows in the table. The getColumnName( ) method returns a column’s name given its ID. The getDownload( ) method takes a row ID and returns the associated Download object from the list. The rest of the DownloadsTableModel class’ methods, which are more involved, are detailed in the following sections.

The addDownload( ) Method

The addDownload( ) method, shown here, adds a new Download object to the list of managed downloads and consequently a row to the table:

```java
// Add a new download to the table.
public void addDownload(Download download) {
    // Register to be notified when the download changes.
    download.addObserver(this);

    downloadList.add(download);

    // Fire table row insertion notification to table.
    fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
}
```

This method first registers itself with the new Download as an Observer interested in receiving change notifications. Next, the Download is added to the internal list of downloads being managed. Finally, a table row insertion event notification is fired to alert the table that a new row has been added.

The clearDownload( ) Method

The clearDownload( ) method, shown next, removes a Download from the list of managed downloads:
// Remove a download from the list.
public void clearDownload(int row) {
    downloadList.remove(row);

    // Fire table row deletion notification to table.
    fireTableRowsDeleted(row, row);
}

After removing the Download from the internal list, a table row deleted event notification is fired to alert the table that a row has been deleted.

The getColumnClass( ) Method

The getColumnClass( ) method, shown here, returns the class type for the data displayed in the specified column:

    // Get a column's class.
    public Class<?> getColumnClass(int col) {
        return columnClasses[col];
    }

All columns are displayed as text (that is, String objects) except for the Progress column, which is displayed as a progress bar (which is an object of type JProgressBar).

The getValueAt( ) Method

The getValueAt( ) method, shown next, is called to get the current value that should be displayed for each of the table’s cells:

    // Get value for a specific row and column combination.
    public Object getValueAt(int row, int col) {
        Download download = downloadList.get(row);
        switch (col) {
            case 0: // URL
                return download.getUrl();
            case 1: // Size
                int size = download.getSize();
                return (size == -1) ? "" : Integer.toString(size);
            case 2: // Progress
                return new Float(download.getProgress());
            case 3: // Status
                return Download.STATUSUSES[download.getStatus()];
        }
        return "";
    }

This method first looks up the Download corresponding to the row specified. Next, the column
specified is used to determine which one of the Download's property values to return.

The update( ) Method

The update( ) method is shown here. It fulfills the Observer interface contract allowing the DownloadsTableModel class to receive notifications from Download objects when they change.

```java
/* Update is called when a Download notifies its observers of any changes. */
public void update(Observable o, Object arg) {
    int index = downloadList.indexOf(o);
    // Fire table row update notification to table.
    fireTableRowsUpdated(index, index);
}
```

This method is passed a reference to the Download that has changed, in the form of an Observable object. Next, an index to that download is looked up in the list of downloads, and that index is then used to fire a table row update event notification, which alerts the table that the given row has been updated. The table will then rerender the row with the given index, reflecting its new values.

The DownloadManager Class

Now that the foundation has been laid by explaining each of the Download Manager's helper classes, we can look closely at the DownloadManager class. The DownloadManager class is responsible for creating and running the Download Manager's GUI. This class has a main( ) method declared, so on execution it will be invoked first. The main( ) method instantiates a new DownloadManager class instance and then calls its show( ) method, which causes it to be displayed.

The DownloadManager class is shown here. Notice that it extends JFrame and implements Observer. The following sections examine it in detail.
/* If not in the middle of clearing a download, 
set the selected download and register to 
receive notifications from it. */
if (!clearing && table.getSelectedRow() > -1) {
    selectedDownload = 
        tableModel.getDownload(table.getSelectedRow());
    selectedDownload.addObserver(DownloadManager.this);
    updateButtons();
}

// Pause the selected download.
private void actionPause() {
    selectedDownload.pause();
    updateButtons();
}

// Resume the selected download.
private void actionResume() {
    selectedDownload.resume();
    updateButtons();
}
// Cancel the selected download.
private void actionCancel() {
    selectedDownload.cancel();
    updateButtons();
}

// Clear the selected download.
private void actionClear() {
    clearing = true;
    tableModel.clearDownload(table.getSelectedRow());
    clearing = false;
    selectedDownload = null;
    updateButtons();
}

/* Update each button's state based off of the currently selected download's status. */
private void updateButtons() {
    if (selectedDownload != null) {
        int status = selectedDownload.getStatus();
        switch (status) {
            case Download.DOWNLOADING:
                pauseButton.setEnabled(true);
                resumeButton.setEnabled(false);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(false);
                break;
            case Download.PAUSED:
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(true);
                break;

            default:
                pauseButton.setEnabled(true);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(true);
                break;
        }
    }
}

public void windowClosing(WindowEvent e) {
    actionExit();
}
// Set up file menu.
JMenuBar menuBar = new JMenuBar();
JMenu fileMenu = new JMenu("File");
fileMenu.setMnemonic(KeyEvent.VK_F);
JMenuItem fileExitMenu = new JMenuItem("Exit",
    KeyEvent.VK_X);
fileExitMenuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        actionExit();
    }
});
fileMenu.add(fileExitMenuItem);
menuBar.add(fileMenu);
setJMenuBar(menuBar);

// Set up add panel.
JPanel addPanel = new JPanel();
addTextField = new JTextField(30);
addPanel.add(addTextField);
JButton addButton = new JButton("Add Download");
addButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        actionAdd();
    }
});
addPanel.add(addButton);

// Set up Downloads table.
tableModel = new DownloadsTableModel();
table = new JTable(tableModel);
table.getSelectionModel().addListSelectionListener(new
    ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        tableSelectionChanged();
    }
});
// Allow only one row at a time to be selected.
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Set up ProgressBar as renderer for progress column.
ProgressBar renderer = new ProgressRenderer(0, 100);
renderer.setStringPainted(true); // show progress text
renderer = new DefaultRenderer(JProgressBar.class, renderer);

// Set table's row height large enough to fit JProgressBar.
table.setRowHeight((int) renderer.getPreferredSize().getHeight());

// Set up downloads panel.
JPanel downloadsPanel = new JPanel();
downloadsPanel.setBorder(
    BorderFactory.createTitledBorder("Downloads"));
downloadsPanel.setLayout(new BorderLayout());
downloadsPanel.add(new JScrollPane(table), BorderLayout.CENTER);

// Set up buttons panel.
JPanel buttonsPanel = new JPanel();
pauseButton = new JButton("Pause");
pauseButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        actionPause();
    }
});
pauseButton.setEnabled(false);
buttonsPanel.add(pauseButton);
resumeButton = new JButton("Resume");
resumeButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        actionResume();
    }
});
resumeButton.setEnabled(false);
buttonsPanel.add(resumeButton);
cancelButton = new JButton("Cancel");
cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        actionCancel();
    }
});
cancelButton.setEnabled(false);
buttonsPanel.add(cancelButton);
clearButton = new JButton("Clear");
clearButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        actionClear();
    }
});
clearButton.setEnabled(false);
buttonsPanel.add(clearButton);

// Add panels to display.
getContentPane().setLayout(new BorderLayout());
getContentPane().add(addPanel, BorderLayout.NORTH);
getContentPane().add(downloadsPanel, BorderLayout.CENTER);
getContentPane().add(buttonsPanel, BorderLayout.SOUTH);

// Exit this program.
private void actionExit() {
    System.exit(0);
}

// Add a new download.
private void actionAdd() {
    URL verifiedUrl = verifyUrl(addTextField.getText());
    if (verifiedUrl != null) {
        tableModel.addDownload(new Download(verifiedUrl));
        addTextField.setText(""); // reset add text field
    } else {
        JOptionPane.showMessageDialog(this,
        "Invalid Download URL", "Error",
        JOptionPane.ERROR_MESSAGE);
    }
}

// Verify download URL.
private URL verifyUrl(String url) {
    // Only allow HTTP URLs.
    if (!url.toLowerCase().startsWith("http://"))
        return null;

    // Verify format of URL.
    URL verifiedUrl = null;
    try {
        verifiedUrl = new URL(url);
    } catch (Exception e) {
        return null;
    }
// Make sure URL specifies a file.
if (verifiedUrl.getFile().length() < 2)
    return null;

return verifiedUrl;

// Called when table row selection changes.
private void tableSelectionChanged() {
    /* Unregister from receiving notifications
    from the last selected download. */
    if (selectedDownload != null)
        selectedDownload.deleteObserver(DownloadManager.this);

    /* If not in the middle of clearing a download,
    set the selected download and register to
    receive notifications from it. */
    if (!clearing && table.getSelectedRow() > -1) {
        selectedDownload =
            tableModel.getDownload(table.getSelectedRow());
        selectedDownload.addObserver(DownloadManager.this);
        updateButtons();
    }
}

// Pause the selected download.
private void actionPause() {
    selectedDownload.pause();
    updateButtons();
}

// Resume the selected download.
private void actionResume() {
    selectedDownload.resume();
    updateButtons();
}

// Cancel the selected download.
private void actionCancel() {
    selectedDownload.cancel();
    updateButtons();
}

// Clear the selected download.
private void actionClear() {
    clearing = true;
    tableModel.clearDownload(table.getSelectedRow());
    clearing = false;
    selectedDownload = null;
    updateButtons();
}
private void updateButtons() {
    if (selectedDownload != null) {
        int status = selectedDownload.getStatus();
        switch (status) {
            case Download.DOWNLOADING:
                pauseButton.setEnabled(true);
                resumeButton.setEnabled(false);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(false);
                break;
            case Download.PAUSED:
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(false);
                break;
            case Download.ERROR:
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(false);
                clearButton.setEnabled(true);
                break;
            default: // COMPLETE or CANCELLED
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(false);
                cancelButton.setEnabled(false);
                clearButton.setEnabled(true);
                break;
        }
    } else {
        // No download is selected in table.
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(false);
    }
}
The DownloadManager Variables
DownloadManager starts off by declaring several instance variables, most of which hold references to the GUI controls. The selectedDownload variable holds a reference to the Download object represented by the selected row in the table. Finally, the clearing instance variable is a boolean flag that tracks whether or not a download is currently being cleared from the Downloads table.

The DownloadManager Constructor
When the DownloadManager is instantiated, all of the GUI's controls are initialized inside its constructor. The constructor contains a lot of code, but most of it is straightforward. The following discussion gives an overview.

First, the window’s title is set with a call to setTitle( ). Next, the setSize( ) call establishes the window’s width and height in pixels. After that, a window listener is added by calling addWindowListener( ), passing a WindowAdapter object that overrides the windowClosing( ) event handler. This handler calls the
actionExit( ) method when the application’s window is closed. Next, a menu bar with a “File” menu is added to the application’s window. Then the “add” panel, which has the text field and button, is set up. An ActionListener is added to the “Add Download” button so that the actionAdd( ) method is called each time the button is clicked.

The downloads table is constructed next. A ListSelectionListener is added to the table so that each time a row is selected in the table, the tableSelectionChanged( ) method is invoked. The table’s selection mode is also updated to ListSelectionModel.SINGLE_SELECTION so that only one row at a time can be selected in the table. Limiting row selection to only one row at a time simplifies the logic for determining which buttons should be enabled in the GUI when a row in the download table is selected. Next, a ProgressRenderer class is instantiated and registered with the table to handle the “Progress” column. The table’s row height is updated to the ProgressRenderer’s height by calling table.setRowHeight( ). After the table has been assembled and tweaked, it is wrapped in a JScrollPane to make it scrollable and then added to a panel.

Finally, the buttons panel is created. The buttons panel has Pause, Resume, Cancel, and Clear buttons. Each of the buttons adds an ActionListener that invokes its respective action method when it is clicked. After creating the buttons panel, all of the panels that have been created are added to the window.

The verifyUrl( ) Method

The verifyUrl( ) method is called by the actionAdd( ) method each time a download is added to the Download Manager. The verifyUrl( ) method is shown here:

```java
// Verify download URL.
```
private URL verifyUrl(String url) {
    // Only allow HTTP URLs.
    if (!url.toLowerCase().startsWith("http://"))
        return null;

    // Verify format of URL.
    URL verifiedUrl = null;
    try {
        verifiedUrl = new URL(url);
    } catch (Exception e) {
        return null;
    }

    // Make sure URL specifies a file.
    if (verifiedUrl.getFile().length() < 2)
        return null;
    return verifiedUrl;
}

This method first verifies that the URL entered is an HTTP URL since only HTTP is supported. Next, the URL being verified is used to construct a new URL class instance. If the URL is malformed, the URL class constructor will throw an exception. Finally, this method verifies that a file is actually specified in the URL.

The tableSelectionChanged() Method

The tableSelectionChanged() method, shown here, is called each time a row is selected in the downloads table:

```java
// Called when table row selection changes.
private void tableSelectionChanged() {
    /* Unregister from receiving notifications from the last selected download. */
    if (selectedDownload != null)
        selectedDownload.deleteObserver(DownloadManager.this);

    /* If not in the middle of clearing a download,
    set the selected download and register to receive notifications from it. */
    if (!clearing && table.getSelectedRow() > -1) {
        selectedDownload =
            tableModel.getDownload(table.getSelectedRow());
        selectedDownload.addObserver(DownloadManager.this);
        updateButtons();
    }
}
```

This method starts by seeing if there is already a row currently selected by checking if the selectedDownload
variable is null. If the selectedDownload variable is not null, DownloadManager removes itself as an observer of the download so that it no longer receives change notifications. Next the clearing flag is checked. If the table is not empty and the clearing flag is false, then first the selectedDownload variable is updated with the Download corresponding to the row selected. Second, the DownloadManager is registered as an Observer with the newly selected Download. Finally, updateButtons( ) is called to update the button states based on the selected Download’s state.

The updateButtons( ) Method

The updateButtons( ) method updates the state of all the buttons on the button panel based on the state of the selected download. The updateButtons( ) method is shown here:
private void updateButtons() {
    if (selectedDownload != null) {
        int status = selectedDownload.getStatus();
        switch (status) {
            case Download.DOWNLOADING:
                pauseButton.setEnabled(true);
                resumeButton.setEnabled(false);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(false);
                break;
            case Download.PAUSED:
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(true);
                clearButton.setEnabled(false);
                break;
            case Download.ERROR:
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(true);
                cancelButton.setEnabled(false);
                clearButton.setEnabled(true);
                break;
            default: // COMPLETE or CANCELLED
                pauseButton.setEnabled(false);
                resumeButton.setEnabled(false);
                cancelButton.setEnabled(false);
                clearButton.setEnabled(true);
                break;
        }
    } else {
        // No download is selected in table.
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(false);
    }
}

If no download is selected in the downloads table, all of the buttons are disabled, giving them a grayed-out appearance. However, if there is a selected download, each button’s state will be set based on whether the Download object has a status of DOWNLOADING, PAUSED, ERROR, COMPLETE, or CANCELLED.

Handling Action Events

Each of DownloadManager’s GUI controls registers an ActionListener that invokes its respective action method. ActionListeners are triggered each time an action event takes place on a GUI control. For example, when a
button is clicked, an ActionEvent is generated and each of the button’s registered ActionListeners is notified. You may have noticed a similarity between the way ActionListeners work and the Observer pattern discussed earlier. That is because they are the same pattern with two different naming schemes.

Compiling and Running the Download Manager

Compile DownloadManager like this:

```
javac DownloadManager.java DownloadsTableModel.java ProgressRenderer.java Download.java
```

Run DownloadManager like this:

```
javaw DownloadManager
```

The Download Manager is easy to use. First, enter the URL of a file that you want to download in the text field at the top of the screen. For example, to download a file called 0072229713_code.zip from the McGraw-Hill web site enter

```
```

This is the file that contains the code for my book The Art of Java, which I co-authored with James Holmes.

After adding a download to the Download Manager, you can manage it by selecting it in the table. Once selected, you can pause, cancel, resume, and clear a download. Figure 34-2 shows the Download Manager in action.

Enhancing the Download Manager

The Download Manager as it stands is fully functional, with the ability to pause and resume downloads as well as download multiple files at once; however, there are several enhancements that you may want to try on your own. Here are some ideas: proxy server support, FTP and HTTPS support, and drag-and-drop support. A particularly appealing enhancement is a scheduling
feature that lets you schedule a download at a specific time, perhaps in the middle of the night when system resources are plentiful.

Note that the techniques illustrated in this chapter are not limited to downloading files in the typical sense. There are many other practical uses for the code. For example, many software programs distributed over the Internet come in two pieces. The first piece is a small, compact application that can be downloaded quickly. This small application contains a mini download manager for downloading the second piece, which is generally much larger. This concept is quite useful, especially as the size of applications increases, which typically leads to an increase in the potential for download interruptions. You might want to try adapting the Download Manager for this purpose.

Figure 34-2 The Download Manager in action
As explained in Part I, Java supports three types of comments. The first two are the // and the /* */. The third type is called a documentation comment. It begins with the character sequence /**. It ends with */. Documentation comments allow you to embed information about your program into the program itself. You can then use the javadoc utility program (supplied with the JDK) to extract the information and put it into an HTML file. Documentation comments make it convenient to document your programs. You have almost certainly seen documentation generated with javadoc, because that is the way the Java API library was documented.

The javadoc Tags
The javadoc utility recognizes the following tags:
Document tags that begin with an “at” sign (@) are called stand-alone tags (also called block tags), and they must be used on their own line. Tags that begin with a brace, such as {@code}, are called in-line tags, and they can be used within a larger description. You may also use other, standard HTML tags in a documentation comment. However, some tags, such as headings, should not be used because they disrupt the look of the HTML file produced by javadoc.

As it relates to documenting source code, you can use documentation comments to document classes, interfaces,
fields, constructors, and methods. In all cases, the documentation comment must immediately precede the item being documented. Some tags, such as @see, @since, and @deprecated, can be used to document any element. Other tags apply only to the relevant elements. Each tag is examined next.

NOTE

Documentation comments can also be used for documenting a package and preparing an overview, but the procedures differ from those used to document source code. See the javadoc documentation for details on these uses.

@author

The @author tag documents the author of a class or interface. It has the following syntax:

```
@author description
```

Here, description will usually be the name of the author. You will need to specify the -author option when executing javadoc in order for the @author field to be included in the HTML documentation.

{@code}

The {@code} tag enables you to embed text, such as a snippet of code, into a comment. That text is then displayed as is in code font, without any further
processing, such as HTML rendering. It has the following syntax:

```{@code code-snippet}
```

@deprecated

The `@deprecated` tag specifies that a program element is deprecated. It is recommended that you include `@see` or `@link` tags to inform the programmer about available alternatives. The syntax is the following:

```@deprecated description```

Here, `description` is the message that describes the deprecation. The `@deprecated` tag can be used in documentation for fields, methods, constructors, classes, and interfaces.

```{@docRoot}```

`{@docRoot}` specifies the path to the root directory of the current documentation.

@exception

The `@exception` tag describes an exception to a method. It has the following syntax:

```@exception exception-name explanation```

Here, the fully qualified name of the exception is specified by `exception-name`, and `explanation` is a string
that describes how the exception can occur. The
@exception tag can only be used in documentation for a
method or constructor.

{@inheritDoc}

This tag inherits a comment from the immediate
superclass.

{@link}
The {@link} tag provides an in-line link to additional
information. It has the following syntax:

    {@link pkg.class#member text}

Here, pkg.class#member specifies the name of a class or
method to which a link is added, and text is the string
that is displayed.

{@linkplain}

Inserts an in-line link to another topic. The link is
displayed in plain-text font. Otherwise, it is similar to
{@link}.

{@literal}
The {@literal} tag enables you to embed text into a
comment. That text is then displayed as is, without any
further processing, such as HTML rendering. It has the
following syntax:

    {@literal description}
Here, description is the text that is embedded.

@param

The @param tag documents a parameter. It has the following syntax:

    @param parameter-name explanation

Here, parameter-name specifies the name of a parameter. The meaning of that parameter is described by explanation. The @param tag can be used only in documentation for a method or constructor, or a generic class or interface.

@return

The @return tag describes the return value of a method. It has the following syntax:

    @return explanation

Here, explanation describes the type and meaning of the value returned by a method. The @return tag can be used only in documentation for a method.

@see

The @see tag provides a reference to additional information. Its most commonly used forms are shown here:

    @see anchor
In the first form, anchor is a link to an absolute or relative URL. In the second form, `pkg.class#member` specifies the name of the item, and `text` is the text displayed for that item. The `text` parameter is optional, and if not used, then the item specified by `pkg.class#member` is displayed. The member name, too, is optional. Thus, you can specify a reference to a package, class, or interface in addition to a reference to a specific method or field. The name can be fully qualified or partially qualified. However, the dot that precedes the member name (if it exists) must be replaced by a hash character.

@serial

The `@serial` tag defines the comment for a default serializable field. It has the following syntax:

    @serial description

Here, `description` is the comment for that field.

@serialData

The `@serialData` tag documents the data written by the `writeObject()` and `writeExternal()` methods. It has the following syntax:

    @serialData description
Here, description is the comment for that data.

@serialField

For a class that implements Serializable, the @serialField tag provides comments for an ObjectStreamField component. It has the following syntax:

    @serialField name type description

Here, name is the name of the field, type is its type, and description is the comment for that field.

@since

The @since tag states that an element was introduced in a specific release. It has the following syntax:

    @since release

Here, release is a string that designates the release or version in which this feature became available.

@throws

The @throws tag has the same meaning as the @exception tag.

{valueOf}

{valueOf} has two forms. The first displays the value of the constant that it precedes, which must be a static field. It has this form:

    {valueOf}
The second form displays the value of a specified static field. It has this form:

```java
{@value pkg.class#field}
```

Here, `pkg.class#field` specifies the name of the static field.

@version

The `@version` tag specifies the version of a class or interface. It has the following syntax:

```java
@version info
```

Here, `info` is a string that contains version information, typically a version number, such as 2.2. You will need to specify the `-version` option when executing javadoc in order for the `@version` field to be included in the HTML documentation.

The General Form of a Documentation Comment

After the beginning `/**`, the first line or lines become the main description of your class, interface, field, constructor, or method. After that, you can include one or more of the various `@` tags. Each `@` tag must start at the beginning of a new line or follow one or more asterisks (`*`) that are at the start of a line. Multiple tags of the same type should be grouped together. For example, if you have three `@see` tags, put them one after the
In-line tags (those that begin with a brace) can be used within any description.

Here is an example of a documentation comment for a class:

```java
/**
 * This class draws a bar chart.
 * @author Herbert Schildt
 * @version 3.2
 */
```

What javadoc Outputs

The javadoc program takes as input your Java program’s source file and outputs several HTML files that contain the program’s documentation. Information about each class will be in its own HTML file. javadoc will also output an index and a hierarchy tree. Other HTML files can be generated.

An Example that Uses Documentation Comments

Following is a sample program that uses documentation comments. Notice the way each comment immediately precedes the item that it describes. After being processed by javadoc, the documentation about the SquareNum class will be found in SquareNum.html.
import java.io.*;
/**
 * This class demonstrates documentation comments.
 * @author Herbert Schildt
 * @version 1.2
 */

public class SquareNum {
    /**
     * This method returns the square of num.
     * This is a multiline description. You can use
     * as many lines as you like.
     * @param num The value to be squared.
     * @return num squared.
     */
    public double square(double num) {
        return num * num;
    }

    /**
     * This method inputs a number from the user.
     * @return The value input as a double.
     * @exception IOException On input error.
     * @see IOException
     */
    public double getNumber() throws IOException {
        // create a BufferedReader using System.in
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader inData = new BufferedReader(isr);
        String str;

        str = inData.readLine();
        return (new Double(str)).doubleValue();
    }
}
public static void main(String args[])
    throws IOException
{
    SquareNum ob = new SquareNum();
    double val;
    System.out.println("Enter value to be squared: ");
    val = ob.getNumber();
    val = ob.square(val);

    System.out.println("Squared value is "+ val);
}
Index

&

bitwise AND, 66, 67, 68–69
Boolean logical AND, 75–76
and bounded type declarations, 336

&& (short-circuit AND), 75, 76–77

*

multiplication operator, 28, 61–62
regular expression quantifier, 913
used in import statement, 194, 321

@

annotation syntax, 276
used with tags (javadoc), 1080, 1083

|

bitwise OR, 66, 67, 68–69
Boolean logical OR, 75–76

|| (short-circuit OR), 75, 76–77
character class specification, 913, 917

bitwise exclusive OR, 66, 67, 68–69

Boolean logical exclusive OR, 75–76

: (used with a label), 105

, (comma), 33, 95

format flag, 557, 559

{ }, 24, 25, 26, 30, 33, 45, 53, 56, 81, 82, 89, 211

used with javadoc tags, 1080

=, 27, 74, 77

= = (Boolean logical operator), 75

= = (relational operator), 28, 74, 260, 266

versus equals( ), 380–381

!, 75–76

! =, 74, 75

/, 61–62
diamond operator (type inference), 359–360
and generic type parameter, 328, 334, 346
< ?, >, 277, 279
< <, 66, 69–70
< =, 74
–, 61, 62
    format flag, 557, 558
– –, 30, 61, 64–65
%
    used in format conversion specifier syntax, 550
modulus operator, 61, 63
( format flag, 557, 559
( ), 25, 33, 79, 114, 123

used to raise the precedence of operations, 33, 79, 376

.

dot operator, 111, 117–118, 146, 170
in multileveled package statement, 188
in import statement, 194
regular expression wildcard character, 913, 916
separator, 33

... (variable-length argument syntax), 156, 159
+

addition operator, 61–62
concatenation operator, 27, 152–153, 374–376
regular expression quantifier, 913, 915–916
unary plus, 61, 62

+ +, 30, 61, 64–66

# format flag, 557, 559

?
regular expression quantifier, **913, 916**
wildcard argument specifier, **338, 341, 344, 367**

`?:` (ternary if-then-else operator), **75, 77–78**

`>`, **28, 74**

`>>`, **66, 70–72**

`>>>`, **66, 72–73**

`>=`, **74**

`;` (semicolon), **26, 33, 90**

used in try-with-resources statement, **305, 589**

`~` (bitwise unary NOT operator), **66, 67, 68–69**

`(underscore)`, **42, 43**

A

`abs( )`, **131–132, 434**

abstract type modifier, **182, 185, 199**

Abstract Window Toolkit. See AWT (Abstract Window Toolkit)

AbstractButton class, **969, 971, 972**
AbstractCollection class, 465, 467, 474
AbstractList class, 465, 466, 509
AbstractMap class, 487, 488, 489, 491
AbstractQueue class, 465, 473
AbstractSequentialList class, 465, 469
AbstractSet class, 465, 470, 472, 475
accept( ), 586, 587, 656, 682
Access control, 141–144
  example program, 191–194
  and inheritance, 142, 144, 163–164
  and packages, 187, 190–194
Access modifiers, 25, 142, 190–191
acos( ), 433
acquire( ), 864–867
ActionEvent class, 710, 711, 776, 777, 787, 812, 956, 967, 969, 975
ActionListener interface, 720, 721, 776, 787, 812, 956,
actionPerformed( ), adapt( ), 
Adapter classes, 
add( ), addAll( ), addCookie( ), addElement( ), addFirst( ), addImage( ), addItem( ), addKeyListener( ), addLast( ), addMouseListener( ), 
addActionListener( ),
addMouseListener( ), 709, 726
Address, Internet, 668
addSuppressed( ), 222
addTab( ), 977
addTListener( ) 936
addTypeListener( ), 708, 709
AdjustmentEvent class, 710, 711–712, 790
AdjustmentListener interface, 720, 721, 790
adjustmentValueChanged( ), 721
Algorithms, collection, 454–455, 495–500, 508
ALIGN, 700
allocate( ), 631, 641, 660, 663
ALT, 700
anchor constraint field, 806, 807–808
AND operator
  bitwise (&), 66, 67, 68–69
  Boolean logical (&), 75–76
and bounded type declarations, 336

short-circuit (&&), 75, 76–77

Android programming, 16

Animation, cell, 857–860

AnnotatedElement interface, 282, 284, 451

Annotation interface, 276, 282, 450

Annotation(s), 13, 14, 275–288

   built-in, 286–288

   declaration example, 276

   marker, 284–285

   member, default value for, 282–284

   obtaining all, 281–282

   reflection to obtain, using, 277–282

   restrictions on, 288

   retention policy for, specifying, 276–277

   single-member, 285–286

annotationType( ), 276
Annuity for a given investment, maximum
applet for finding, 1040–1044
formula to compute, 1040

Annuity, initial investment needed for desired
applet for finding, 1036–1040
formula to compute, 1036

Apache Software Foundation, 994

API packages, table of core Java, 909–911

APPEND, 421

append( ), 391–392, 449, 611, 795

Appendable interface, 449, 551, 605, 610, 619
appendCodePoint( ), 394
appendTo( ), 421

Applet, 8, 16, 307–309, 1019
architecture, 690–691, 696
basics, 687–690
colors, setting and obtaining, 694–695
examples for financial calculations, 1020–1047
executing, 308–309, 687–688, 699–701
and the Internet, 8–9
and main( ), 26, 110, 308, 309, 688
outputting to console, 706
passing parameters to, 701–703
request for repaint, 695–698
skeleton, 691–693
and socket connections, 671
as source and listener for events, 726
string output to, 308, 688, 694, 696
Swing, 687, 688, 949, 957–959
viewer, 308–309, 687, 699, 739
Applet class, 307, 687–706, 719, 726, 739, 830, 832, 957, 959
methods, table of, 689–690
applet package, 289, 307, 687
APPLET tag, HTML, 308, 309, 688
  full syntax for, 699–701
AppletContext interface, 687, 700, 704–706
  methods, table of, 705
AppletStub interface, 687, 706
appletviewer, 308, 687
  status window, using, 699
Application launcher (java), 24, 188, 189
  and main( ), 25
ARCHIVE, 701
AreaAveragingScaleFilter class, 844
areFieldsSet, 531
Argument(s), 116, 120
  command-line, 25, 154–155
  index, 560–563
  passing, 136–138
  type. See Type argument(s)
variable-length. See Varargs

wildcard. See Wildcard arguments

Arithmetic operators, 61–66

ArithmeticException, 209, 210, 220

Array class, 451

Array(s), 25, 51–58, 147, 185

boundary checks, 53

certaining collections into, 459, 468–469

copying with arraycopy( ), 423, 425–426

declaration syntax, alternative, 58

dynamic, 451, 466–469, 474, 509

and the for-each loop, 97–101

and generics, 366–367

implemented as objects, 147

indexes, 52

initializing, 53, 56–57

length instance variable of, 147–149
multidimensional, 54–57
one-dimensional, 51–53
of strings, 154
and valueOf( ), 386–387
and varargs, 156
ArrayBlockingQueue class, 889
arraycopy( ), 423, 425–426
ArrayDeque class, 465, 474–475, 515
ArrayIndexOutOfBoundsException, 213, 220, 502, 503, 504
ArrayList class, 465, 466–469, 482, 509, 511
pre-generics versus generics version of the, 505–507
Arrays class, 501–505
ArrayStoreException, 220, 501, 502, 503
arrive( ), 876
arriveAndAwaitAdvance( ), 876, 879, 882
arriveAndDeregister( ), 876, 879
ASCII character set, 39, 40, 43
and strings on the Internet, 373, 378

asin( ), 433

asList( ), 501

Assembly language, 4, 5

assert statement, 13, 316–319

Assertion, 316–319

AssertionError, 316

Assignment operator(s)

  =, 27, 74, 77

  arithmetic compound (op =), 61, 63–64

  bitwise compound, 66, 73–74

  Boolean logical, 75

atan( ), 433

atan2( ), 433

Atomic operations, 892–893

AtomicInteger class, 863, 892–893
AtomicLong class, 863, 892
AttributeView interface, 639
AudioClip interface, 687, 706
Autoboxing/unboxing, 14, 268, 270–275, 329, 330
Boolean and Character values, 274
and the Collections Framework, 456, 469
definition of, 270
and error prevention, 274–275
and expressions, 272–273
and methods, 271–272
when to use, 275
Automatic resource management (ARM), 208, 303–306, 449, 562, 674
available( ), 591, 592–594, 625, 626
availableProcessors( ), 903
await( ), 869–871, 872, 890

awaitAdvance( ), 882

awaitInterruptibly( ), 882

AWT (Abstract Window Toolkit), 289, 307, 309, 687, 688, 773

and applet architectural constraints, 696

classes, table of some, 736–737

color system, 755

controls. See Controls, AWT

creating stand-alone windows with, 747–748

layout managers. See Layout manager(s)
support for imaging, 829

support for text and graphics, 749

and Swing, 735, 945, 946

AWTEvent class, 710, 736

bit mask constants, 822–823
B, 4

BASIC, 4

BasicFileAttributes class, 638–639, 652

    methods, table of, 638

BasicFileAttributeView interface, 639

Basic multilingual plane (BMP), 414

BCP 47, 539

BCPL, 4

BeanInfo interface, 934, 936–937

Beans, Java. See Java Beans

Bell curve, 540

Bell Laboratories, 6

Berkeley UNIX, 667

Berners-Lee, Tim, 675

Beyond Photography, The Digital Darkroom (Holzmann), 840

Binary
literals, 42

numbers and integers, 66–67

binarySearch( ), 495, 496, 501

BitSet class, 527–530

   methods, table of, 527–528

Bitwise operators, 66–74

BLOCKED, 256

Blocks of code. See Code blocks

Boolean, 35

   literals, 43

   logical operators, 75–77

Boolean class, 269, 274, 414–416

   methods, table of, 416

boolean data type, 35, 40–41, 43, 48

   and relational operators, 74–75

booleanValue( ), 269, 416

Border interface, 964
BorderFactory class, 964

BorderLayout class, 736, 798–800, 956

Boxing, 270

break statement, 84–87, 102–106

and the for-each loop, 98–99

as form of goto, 104–106

Buffer class, 630–631

methods, table of, 630–631

Buffer, NIO, 630–631

BufferedInputStream class, 291, 599–601, 651

BufferedOutputStream class, 291, 599, 601–602, 651

BufferedReader class, 292, 293, 294–295, 616–618

BufferedWriter class, 292, 619

Buffering, double, 833–836

bulkRegister( ), 882

Button class, 736, 776

extending, 823–824
ButtonGroup class, 965, 975

ButtonModel interface, 948, 969

Buttons, 720
  push. See Push buttons
  radio. See Radio buttons
  Swing, 956–957, 969–977

ButtonUI, 948

Byte class, 269, 403, 410
  methods defined by, table of, 404

byte data type, 35, 36–37, 41

ByteArrayInputStream class, 291, 596–597

ByteArrayOutputStream class, 291, 598–599

ByteBuffer class, 631, 640, 641, 644, 660
  get( ) and put( ) methods, table of, 632

Bytecode, 9–10, 12, 13, 16, 23–24, 312–313, 323, 436

byteValue( ), 269, 398, 399, 400, 404, 405, 406, 408
C
  history of, 4–5
  and Java, 3, 5, 7, 11

C Programming Language, The (Kernighan and Ritchie), 4

C++
  history of, 5–6
  and Java, 3, 7, 11

C# and Java, 8

Calendar class, 530, 531–534, 535, 539
  constants, 533
  methods defined by, table of commonly used, 532–533

Call-by-reference, 136, 137–138

Call-by-value, 136–137, 138

call( ), 885, 907

Callable interface, 863, 885, 886–887, 907
CallSite class, 451

cancel( ), 545, 546, 906

Canvas class, 736, 739, 830

capacity( ), 389–390, 510, 630

capacityIncrement Vector data member, 509

Card layouts, 802–805

CardLayout class, 736, 802–803

CaretEvent class, 967

Case sensitivity and Java, 23, 25, 32

case statement, 84–87, 88–89

Casts, 48–49, 50, 326, 328, 329, 330, 331, 332

   and casting one instance of a generic class into another, 357–358

   and erasure, 328–329, 361–362

   using instanceof with, 310–312

catch block(s), 207, 208, 210–213, 218

   displaying exception description within, 212
and the more precise (final) rethrow feature, 225, 226

dulti-catch feature of, 225–226

using multiple, 212–213

and nested try statements, 211, 214

cbrt( ), 434

ceil( ), 434

CGI (Common Gateway Interface), 10, 993–994

Channel interface, 631–632

Channel(s), NIO, 630, 631–633

char data type, 35, 39–40, 48, 61

Character class, 269, 274, 411–414

methods, table of various, 413, 415

support for 32-bit Unicode, 414

Character(s), 35, 39–40

basic multilingual plane (BMP), 414

changing case of, 387
classes (regular expressions), 913, 917

code point, 414

escape sequences, 43, 44

extraction from String objects, 377–378

formatting an individual, 551

literals, 43

supplemental, 414

Character.Subset class, 397, 414

Character.UnicodeBlock class, 397, 414

CharArrayReader class, 292, 614–615

CharArrayWriter class, 292, 615–616

charAt(), 153, 377, 390–391, 448

CharBuffer class, 631

CharSequence interface, 371, 392, 395, 448, 912

Charsets, 633

charValue(), 269, 411

Check boxes, 691, 720, 779–783
Swing, 973–975

checkAll( ), 836

Checkbox class, 736, 779–780 extending, 824–825

CheckboxGroup class, 736, 782–783 extending, 825–826

CheckboxMenuItem class, 736, 810, 811, 812

checked... methods, 496, 499

checkedCollection( ), 496, 499

checkedList( ), 496, 499

checkedMap( ), 496, 499

checkedSet( ), 496, 499

checkID( ), 836

Choice class, 736, 783–784 extending, 826

Choice controls, 720, 783–786

Class class, 277–278, 281, 282, 429–433, 639, 919
methods, table of some, 430–431
.class filename extension, 24, 112

class keyword, 24, 109

CLASS retention policy, 276–277

Class(es), 109–128

abstract, 181–184, 185, 199

access levels of, 190–191

adapter, 729–731

character, regular expression, 913, 917

and code, 23, 109, 190

in collections, storing user-defined, 480–481

constructor. See Constructor

controlling access to. See Access control

as a data type, 109, 111, 113, 114, 126

definition of, 19

encapsulation achieved through, 126–127

final, 185
general form of, 109–110

generic. See Generic class

inner. See Inner classes

and interfaces, 196, 197–200

libraries, 23, 34

literal, 279

member. See Member, class

name and source file name, 23, 24

nested, 149, 151

packages as containers for, 187, 190

public, 191

scope defined by a, 45–46

type for bounded types, using a, 335–336

ClassCastException, 220, 457, 459, 461, 463, 483, 484, 485, 492, 499, 501, 503, 504

ClassDefinition class, 450

ClassFileTransformer interface, 450
ClassLoader class, 433

classModifiers( ), 922

ClassNotFoundException, 221, 625

CLASSPATH, 188, 189, 926

–classpath option, 188, 189

ClassValue class, 448

clear( ), 457, 458, 483, 517, 527, 532, 630

Client/server model, 8, 10, 667

and sockets, 671–675

clone( ), 185, 427–429, 447, 510, 517, 527, 530, 532, 536, 1008

Cloneable interface, 427–429

CloneNotSupportedException, 221, 427, 447

Cloning, potential dangers of, 427–429

close( ), 298, 303–306, 450, 549, 562, 564, 569, 573, 588, 589, 591, 592, 596, 598, 608, 609, 611, 614, 623, 624, 625, 626, 672, 674, 683

within a finally block, calling, 300
Closeable interface, 298, 304, 569, 588, 591, 594, 605, 607, 608, 610, 619, 631

COBOL, 4

CODE, 700, 701

Code base, 704

Code blocks, 28, 30–31, 45, 82–83

and the break statement, 104–106

and scopes, 45, 46

static, 145, 313

Code point, definition of, 414

Code, unreachable, 213

CODEBASE, 700

codePointAt( ), 388, 394, 414, 415

codePointBefore( ), 388, 394, 415

codePointCount( ), 388, 394

Collection interface, 456, 457–459

methods defined by, table of, 458
Collection-view, 455, 482, 483–484

Collection(s), 325, 453–524

- algorithms, 454–455, 495–500, 508
- into arrays, converting, 459, 468–469
- and autoboxing, 456, 469
- classes, 465–476
- concurrent, 862, 889
- dynamically typesafe view of a, 499
- and the for-each version of the for loop, 97, 101, 456, 479–480

Framework. See Collections Framework

- generic nature of, 455
- interfaces, 454, 456–465
- and iterators, 455, 456, 476–480
- and legacy classes and interfaces, 508–524
- modifiable versus unmodifiable, 457
- pre-generics, problems with, 505–506
and primitive types, 456, 469
storing user-defined classes in, 480–481
and synchronization, 465, 499, 508
and type safety, 455, 499, 505
when to use, 524
Collections class, 454, 495, 499, 508
algorithms defined by, table of, 495–499
Collections Framework, 13, 97, 101, 270, 453–524
advantages of generics as applied to the, 455, 505–508
JDK 5 changes to, 325, 455–456
overview, 454–455
Color class, 736, 755–757
constants 694
Combo boxes, Swing, 984–986
ComboBoxModel interface, 985
Comment, 24–25
Common Gateway interface (CGI), 10, 993–994
Comparable interface, 348, 381, 448, 530, 585
Comparable <Path> interface, 634
Comparator interface, 457, 489, 492
comparator( ), 461, 474, 484
Comparators, 472, 474, 489, 490, 492–495
compare( ), 399, 400, 404, 405, 406, 408, 416, 492–493
compareAndSet( ), 863, 892
compareTo( ), 265, 266, 381–382, 399, 400, 404, 405, 406, 408, 412, 416, 447, 448, 493, 530, 585
compareToIgnoreCase( ), 382
Compilation unit, 23
compile( ), 911
Compiler class, 436
Compiler, Java, 23–24
and main( ), 25

Component class, 688, 691, 694, 696, 709, 719, 726, 736, 738, 739, 742, 749, 774, 796, 828, 830, 948, 949, 960

ComponentAdapter class, 730

componentAdded( ), 721

ComponentEvent class, 710, 712, 713, 719

componentHidden( ), 721

ComponentListener interface, 720, 721, 730

componentMoved( ), 721

componentRemoved( ), 721

componentResized( ), 721

Components, AWT, 945–946, 948

and overriding paint( ), 828

Components, Swing, 948–949

architecture, 947–948

class names for, table of, 948–949
heavyweight, 949
lightweight, 946, 965
painting, 959–964
componentShown( ), 721
ComponentUI, 948
compute( ), 895, 896, 900, 903, 905, 908
concat( ), 385
Concurrent API, 861–862
    packages, 862–863
Concurrent collection classes, 862, 889
Concurrent program, definition of, 861
ConcurrentHashMap class, 863, 889
ConcurrentLinkedDeque, 889
ConcurrentLinkedQueue class, 863, 889
ConcurrentSkipListMap class, 889
ConcurrentSkipListSet class, 889
Condition class, 890
connect( ), 672
Console class, 620–622
  methods, table of, 621
console( ), 423, 620
const keyword, 33
Constants, 32
Constructor class, 278, 281, 282, 451, 920
Constructor(s), 114, 121–124
  in class hierarchy, order of calling, 174–175
default, 114, 123
enumeration, 263–265
factory methods versus overloaded, 669
generic, 346–347
object parameters for, 135–136
overloading, 132–134
parameterized, 123–124
and super( ), 167–170, 174, 323
this( ) and overloaded, 321–323
constructorModifiers( ), 922
Container class, 688, 736, 739, 774, 796, 799, 800, 948, 949, 961
ContainerAdapter class, 730
ContainerEvent class, 710, 712–713
ContainerListener interface, 720, 721, 730
Container(s), Swing, 948, 949
  lightweight versus heavyweight, 949
  panes, 949
Containment hierarchy, 948, 949
contains( ), 388, 457, 458, 471, 510, 517
containsAll( ), 457, 458
Content pane, 949, 952–953, 957, 963, 977, 980, 987,
default layout manager of JFrame, 953, 956
contentEquals(), 388
Context switching, 227, 257
rules for, 229
continue statement, 106–107
Control statements. See Statements, control
Controls, AWT, 773–795
fundamentals, 773–774
convert(), 888
Convolution filters, 846, 852
Cookie class, 1004, 1007–1008
methods, table of, 1008
CookieHandler class, 681
CookieManager class, 681
CookiePolicy interface, 681
Cookies, 1007–1008
example servlet using, 1013–1015
CookieStore interface, 681
copy( ), 636, 648–649
copyOf( ), 475, 476, 501
copyOfRange( ), 502
CopyOnWriteArrayList class, 863, 889
CopyOnWriteArraySet class, 889
copySign( ), 435
cos( ), 38, 433
cosh( ), 433
countDown( ), 869–871
CountDownLatch class, 862, 869–871
countStackFrames( ), 437
createImage( ), 830, 839, 844
createLineBorder( ), 964
CropImageFilter class, 844–845
Currency class, 547–548
methods, table of, 547–548

`currentThread()`, 231, 438

`currentTimeMillis()`, 423, 425

CyclicBarrier class, 862, 871–873

D

Data types, 27. See also Type(s); Types, primitive

DatagramPacket class, 682, 683–684

Datagrams, 668, 682–685

  server/client example, 684–685

DatagramSocket class, 632, 682–683

DataInput interface, 607, 608, 609, 625

DataInputStream class, 291, 607, 608–609

DataOutput interface, 607, 608, 609, 623

DataOutputStream class, 291, 607–608

Date class, 530–531, 927, 928

  methods, table of, 530

DateFormat class, 530, 539, 927–929
Deadlock, 249–251, 253, 437
Decrement operator (– –), 30, 61, 64–65
decrementAndGet( ), 863, 892
deepequals( ), 502
deepphasHashcode( ), 504
deeptoString( ), 504
default
    clause for annotation member, 282–283
    statement, 84–85
DefaultmutableTreeNode class, 987, 988
DelayQueue class, 889
Delegation event model, 708–709, 710
    and Beans, 936
    and event listeners, 708, 709, 720–723
    and event sources, 708–709, 719–720
    and Swing, 954
using, 723–729
delete operator, 125
delete( ), 393, 584, 636
deleteCharAt( ), 393
deleteOnExit( ), 585
delimiter( ), 572
Delimiters, 525
    Scanner class, 564, 570–572
@Deprecated built-in annotation, 286, 287
Deque interface, 456, 463–465, 469, 474
    methods, table of, 464
descendingIterator( ), 462, 464, 465
destroy( ), 417, 420, 437, 440, 689, 691, 692, 693, 695, 957, 994, 997, 999
Destructors versus finalize( ), 126
Dialog boxes, 816–822
    file, 820–822
Dialog class, 736, 816
Diamond operator (< >), 359–360

Dictionary class, 454, 508, 515–516
  abstract methods, table of, 515

digit( ), 412

Dimension class, 736, 740, 754

Directories as File objects, 583, 585–586
  creating, 588

Directory, listing the contents of a
  using list( ), 585–587
  using listFiles( ), 587
  using NIO, 654–657

Directory tree, obtaining a list of files in a, 657–659

DirectoryStream class, 654

DirectoryStream.Filter interface, 656

dispose( ), 816

DLL (dynamic link library), 314, 315, 316

do-while loop, 90–93
Document base, 704

Document interface, 967

Document/view methodology, 544

@Documented built-in annotation, 286

doDelete( ), 1009, 1010

doGet( ), 1009, 1010, 1011

doHead( ), 1009, 1010

Domain name, 668, 669

Domain Naming Service (DNS), 668

doOptions( ), 1009, 1010

doPost( ), 1009, 1010, 1012

doPut( ), 1009, 1010

DosFileAttributes class, 639, 654

DosFileAttributeView interface, 639

Dot operator (.), 111, 117–118, 146, 170, 188, 194

doTrace( ), 1009, 1010

Double buffering, 833–836
Double class, 269, 398–402
  methods, table of, 400–402
double data type, 35, 38–39, 42
DoubleBuffer class, 631
doubleValue( ), 269, 398, 399, 401, 404, 405, 406, 408
Download Manager, 1053–1077
  compiling and running, 1076–1077
  enhancing, suggestions for, 1077
  overview of, 1054
Downloads, Internet
  operation of, 1054
  resuming interrupted, 1053
drawArc( ), 752–753
drawImage( ), 831, 834, 835
drawLine( ), 749–750, 960
drawOval( ), 751–752
drawPolygon( ), 753–754

drawRect( ), 750–751, 960

drawRoundRect( ), 750–751

drawString( ), 308, 688, 694, 696, 765

Dynamic link library (DLL), 313–314, 315, 316

Dynamic method

  dispatch, 178–181

  lookup, 198

  resolution, 196

E

  E (Math constant), 433

Early binding, 184

echoCharIsSet( ), 793

Eclipse IDE, 994, 995

Edit control, 792

  element( ), 463

  elementAt( ), 510, 511
elementCount Vector data member, 509

elementData[ ] Vector data member, 509

elements( ), 510, 515, 517

ElementType enumeration, 286–287, 450

else, 81–84

empty( ), 513

EMPTY_LIST static variable, 499

EMPTY_MAP static variable, 499

EMPTY_SET static variable, 499

EmptyStackException, 512, 513

enableEvents( ), 822–823, 824, 826, 827

Encapsulation, 5, 18–19, 20, 22–23, 126–127, 167

and access control, 141

and scope rules, 46

end( ), 912

endsWith( ), 380, 634

ensureCapacity( ), 390, 467, 510
entrySet( ), 483–484, 486, 489, 518

eenum, 259, 447, 476, 491

Enum class, 265, 447

    methods, table of, 447

EnumConstantNotPresentException, 220

enumerate( ), 438, 440, 443

Enumeration interface, 508–509, 511–512, 513, 525, 526, 603

Enumeration(s), 14, 259–268, 447, 513

    = = relational operator and, 260, 266

as a class type in Java, 259, 263–265

constants, 259, 260, 263, 264, 265–266

constructor, 263–265

restrictions, 265

values in switch statements, using, 260–261

variable, declaring an, 260

EnumMap class, 487, 491–492
EnumSet class, 465, 475–476

    factory methods, table of, 476

Environment properties, list of, 427


    versus = =, 380–381

equalsIgnoreCase( ), 379

Erasure, 329, 361–365, 366

    and ambiguity errors, 364–365

    bridge methods and, 362–364

err, 292, 293, 423

Error class, 208–209, 217, 224, 620

Errors

    ambiguity, 364–365

    assertions to check for, using, 316–318

    autoboxing/unboxing and prevention of, 274–275
automatic type promotions and compile-time, 50
compile-time versus run-time, 331–332
generics and prevention of, 330–332, 506
raw types and run-time, 351
run-time, 12, 207, 310. See also Exception handling
unreachable code, 108

Event

and applets, 690–691
definition of an, 708
design patterns for an, 936
dispatching thread and Swing, 953–954, 957, 959
driven programs, 707, 953
loop with polling, 228
model, delegation. See Delegation event model
multicasting and unicasting, 936

Event handling, 690, 707–733

and adapter classes, 729–731
event classes, 709–719
by extending AWT components, 708, 822–828
and inner classes, 731–733
keyboard, 726–729
mouse, 723–726
and Swing, 946, 953–957
See also Delegation event model
Event listener interfaces, 720–723
and adapter classes, 729–731
table of commonly used, 720
EventListener interface, 578
EventListenerProxy class, 577
EventObject class, 577, 709–710, 1009
EventSetDescriptor class, 936, 938, 940
Exception class, 208–209, 221, 223, 224
Exception classes and generics, 367
Exception handling, 12, 93, 102, 207–226, 299–300,
block, general form of, 208
and chained exceptions, 13, 224–225
and creating custom exceptions, 221–223
and the default exception handler, 209–210, 216
and the more precise (final) rethrow feature, 225, 226
and suppressed exceptions, 222, 306
and uncaught exceptions, 450
Exceptions, built-in run-time, 208–209, 220
checked, table of, 221
constructors for, 217
unchecked RuntimeException, table of, 220
Exceptions, I/O, 588–589
exchange( ), 873–875
Exchanger class, 862, 873–875
exec( ), 416, 417–418, 420, 421
execute( ), 882, 896, 905

Executor interface, 863, 882, 883

Executors, 863

using, 882–887

ExecutorService interface, 863, 882–883, 885

exists( ), 583, 636, 652

exitValue( ), 417, 420

exp( ), 434

expm1( ), 434

Expressions

and autoboxing/unboxing, 272–273

automatic type promotion in, 49–51

regular. See Regular expressions

extends, 161, 163, 205, 335, 339

and bounded wildcard arguments, 341, 344

Externalizable interface, 623
false, 34, 40, 41, 43, 75, 76

FALSE, 414

FAT file system, 639, 654

Field class, 278, 281, 282, 451, 920

Field, final, 146–147

dfieldModifiers( ), 922

fields array, 531

File attribute(s)

  File to access, using, 582–585

  interfaces, 648–639

NIO to access, using, 639, 652–654

  view interfaces, 639

File class, 563, 582–588, 605, 619, 652

  instance into a Path instance, converting a, 585, 635, 652

  methods, 583–585, 592
File(s) to a buffer, map a, 633, 644, 647, 661–663, 664

close( ) to close a, using, 300–302, 306

I/O, 297–306, 582–588. See also NIO

pointer, 609, 610

source, 23–24, 110

system, accessing the, 640

try-with-resources to automatically close a, using, 303–306

FileChannel class, 632, 633, 641, 644, 645–646, 660

FileDialog class, 736, 820–822

FileFilter interface, 587

FileInputStream class, 291, 297–298, 592–594, 632, 660, 661, 662, 663

FilenameFilter interface, 586–587

FileNotFoundException, 298, 301, 588, 592, 594, 612
FileOutputStream class, 291, 297–298, 302, 594–596, 632, 663

FileReader class, 292, 564, 612

Files class, 582, 633, 635–637, 639, 648, 649, 652, 654, 657

methods, table of a sampling of, 636–637

FileStore class, 630

FileSystem class, 640

FileSystems class, 640

FileVisitor interface, 657–658

FileVisitResult enumeration, 658

FileWriter class, 292, 613–614

fill( ), 497, 503

fillArc( ), 752–753

fillInStackTrace( ), 222

fillOval( ), 751–752

fillPolygon( ), 753–754
fillRect( ), 750–751
fillRoundRect( ), 750–751
FilteredImageSource class, 839, 844
FilterInputStream class, 291, 599, 608
FilterOutputStream class, 291, 599, 607
FilterReader class, 292
FilterWriter class, 292
final, 146–147
to prevent class inheritance, 185
to prevent method overriding, 184
finalize( ), 125–126, 185, 427
finally block, 207, 208, 218–220, 300–301, 589
Financial calculations, applets and servlets for, 1019–1052
find( ), 912, 914–915, 916
findInLine( ), 572
findWithinHorizon( ), 572–573
Finger protocol, 675
first( ), 461, 803
firstElement( ), 510, 511
firstKey( ), 484
flip( ), 630, 647
Float class, 269, 398–400, 402
    methods, table of, 399–400
float data type, 35, 38, 42
Floating-point(s), 35, 38–39
    literals, 42–43
FloatBuffer class, 631
floatValue( ), 269, 398, 399, 401, 404, 405, 406, 408
floor( ), 434, 462
FlowLayout class, 736, 797–798, 956
flush( ), 549, 588, 592, 601, 611, 621, 623, 624
Flushable interface, 588, 591, 594, 605, 607, 610, 619, 620
FocusAdapter class, 730
FocusEvent class, 710, 712, 713
focusGained( ), 721
FocusListener interface, 720, 721, 730
focusLost( ), 721
Font class, 736, 759–760, 761, 762, 764
   methods, table of some, 760
Font(s), 759–772
   creating and selecting, 762–763
   determining available, 760–761
   information, obtaining, 764
   metrics to manage text output, using, 764–772
   terminology used to describe, 765
FontMetrics class, 736, 764–766
   methods, table of some, 765–766
for loop, 29–31, 93–102
   enhanced. See For-each version of the for loop
variations, 96–97

For-each version of the for loop, 14, 93, 97–101

and arrays, 97–101

and the break statement, 98–99

and collections, 97, 101, 456, 479–480

general form, 97

and the Iterable interface, 449, 456, 479

and maps, 482

forceTermination( ), 882

forDigit( ), 412


advantages to using the, 893–894

classes, main, 894–897

tips for using the, 908

and the sequential processing threshold interaction with the level of parallelism, 900–903

Fork/Join Framework tasks, 894
  asynchronous execution of, 905
  cancelling, 906
  completion status of, 906
  and the parallelism level, 896
  restarting, 906
  starting, 896, 905
  and subtasks, 897
    that do not return a result, 894, 895, 903
    that return a result, 894, 895–896, 903
  fork(), 894–895, 903, 905, 907

ForkJoinPool class, 863, 883, 894, 895, 896–897, 900, 903, 905, 907–908
  and work stealing, 897, 907

ForkJoinTask class, 863, 894–895, 896, 897, 907
Format flags, 557–560

Format specifiers, 548, 549–562

argument index with, using an, 560–562
and format flags, 557–560
and specifying minimum field width, 555–556
and specifying precision, 556–557
suffixes for the time and date, table of, 553

table of, 551

uppercase versions of, 560

format( ), 388, 549–550, 607, 619, 621, 927

Formattable interface, 578

FormattableFlags class, 577

Formatter class, 548–562, 606

closing an instance of the, 562
constructors, 548–549
methods, table of, 549

See also Format specifiers
forName( ), 430, 919

FORTRAN, 4, 5

Frame class, 736, 738, 739–740, 741, 742

Frame window (s), 739–748
  creating stand-alone, 747–748
  handling events in, 742–747
  within applet, creating, 741–742

Frank, Ed, 6

freeMemory( ), 418–419

from( ), 421

FTP (File Transfer Protocol), 668, 675, 1054

Future interface, 863, 885–887

G
  Garbage collection, 12, 125, 126, 139, 418–419, 451, 837
  gc( ), 418, 419, 423

Generic class
example program with one type parameter, 326–329

general form, 334

hierarchies, 352–359

and instanceof, 355–357

overriding methods in a, 358–359

and raw types, 349–352

and type inference, 359–361

Generic constructors, 346–347

Generic interfaces, 326, 347–349

and classes, 348–349


Generics, 13, 14, 270, 325–367

and ambiguity errors, 364–365

and arrays, 366–367
and casts, 326, 328, 329

and the Collections Framework, 325, 455, 505–508

and compatibility with pre-generics code, 349–352, 361

and exception classes, 367

restrictions on using, 365–367

type checking and, 329, 330–332, 351

GenericServlet class, 997, 999, 1002, 1008

get(), 459, 460, 470, 483, 488, 515, 517, 527, 532, 638, 640, 681, 886, 888, 892

and buffers, 631, 632, 643, 661

getActionCommand(), 711, 777, 787, 969, 975, 976

getActiveThreadCount(), 908

getAddListenerMethod(), 940

getAddress(), 670, 684

getAdjustable(), 712

getAdjustmentType(), 712, 790
getAlignment(), 775
getAllByName(), 669, 670
getAllFonts(), 761
getAndSet(), 863, 892, 893
getAnnotation(), 278, 282, 430, 444
getAnnotations(), 281–282, 430, 444
getApplet(), 700, 705
g.getAppletContext(), 689, 705
getApplets(), 705
getArrivedParties(), 882
getAscent(), 765, 766
getAttribute, 1000, 1001, 1007, 1015
getAttributeNames(), 1007, 1015
getAudioClip(), 689, 705, 706
getAvailableFontFamilyNames(), 760–761
getBackground(), 694
getBeanInfo(), 940
getBlue( ), 756
getButton(), 717
getByAddress( ), 670
getByName( ), 669
getBytes( ), 378, 594
getCause( ), 222, 224
getChannel( ), 632, 660, 662, 663
getChars( ), 377–378, 391, 613
getChild( ), 713
getClass( ), 185, 186, 277, 427, 429, 432, 921
getClickCount( ), 717
getCodeBase( ), 689, 704
getColor( ), 756
getComponent( ), 712
getConstructor( ), 278, 430
getConstructors( ), 430, 920
getContainer( ), 733
getContentLength( ), 677
getContentLengthLong( ), 677
getContentPane( ), 953, 956
getContents( ), 575
getContentType( ), 677, 1001
getCookies( ), 1005, 1014
data( ), 684
date( ), 677
dateInstance( ), 927
dateTimeInstance( ), 929
getDeclaredAnnotations( ), 282, 430, 444
getDeclaredMethods( ), 430, 921
default( ), 536, 538
descent( ), 765, 766
directionality( ), 414
directory( ), 821
displayCountry( ), 538
getDisplayLanguage( ), 538
getDisplayName( ), 538
getDocumentBase( ), 689, 704
getEchoChar( ), 793
getErrorStream( ), 417
getEventSetDescriptors( ), 936, 943
getExpiration( ), 677
getExponent( ), 435
GetField inner class, 625
getField( ), 278, 431
GetFieldID( ), 315
getFields( ), 431, 920
getFile( ), 821
getFileAttributeView( ), 639
getFiles( ), 822
getFirst( ), 464, 469
getFollowRedirects( ), 679
getFont( ), 760, 764, 765
getForeground( ), 694
getFreeSpace( ), 585
getGraphics( ), 696, 749, 834
getGreen( ), 756
getHeaderField( ), 677
getHeaderFieldKey( ), 677
getHeaderFields( ), 677, 681
getHeight( ), 765, 766, 961
getHostAddress( ), 670
getHostName( ), 671
getIcon( ), 966
getID( ), 438, 536, 710
getImage( ), 689, 705, 830–831
getInetAddress( ), 672, 683
getInitParameter( ), 1000
getInitParameterNames( ), 1000
getInputStream( ), 417, 420, 672, 677, 1001
getInsets( ), 800, 961
getInstance( ), 532, 534, 547
GetIntField( ), 315
getItem( ), 715, 784, 787, 812, 972, 973–974
getItemCount( ), 784, 787
getItemSelectable( ), 715, 787
getKey( ), 487, 489
getKeyChar( ), 716
getKeyCode( ), 716
getLabel( ), 776, 780, 811
getLast( ), 464, 469
getLastModified( ), 677
getLeading( ), 765, 766
getLength( ), 684
getListenerType( ), 940
getLocale( ), 574, 689
getLocalGraphicsEnvironment( ), 761
getLocalHost( ), 669
getLocalizedMessage( ), 222
getLocalPort( ), 672, 683
getLocationOnScreen( ), 717
getMaximum( ), 790
getMessage( ), 217, 222
getMethod( ), 278, 280, 431, 940, 1005
getMethodDescriptors( ), 936
getMethods( ), 431, 920
getMinimum( ), 790
getMinimumSize( ), 796–797
getModifiers( ), 711, 714, 921
getModifiersEx( ), 714
getName( ), 230, 232, 431, 438, 440, 445, 583, 634, 652, 760, 921, 940, 1008, 1010, 1014
getNameCount( ), 634
getNewState( ), 719
GetObjectClass( ), 315
getOffset( ), 536, 684
getOldState( ), 719
getOppositeComponent( ), 713
getOppositeWindow( ), 719
getOutputStream( ), 417, 420, 672, 1001
getParallelism( ), 903
getParameter( ), 689, 701, 702, 1001, 1003, 1011, 1012
getParameterNames( ), 1001, 1003
getParent( ), 440, 583, 634, 652, 882
getPath( ), 987, 1008
getPhase( ), 877
getPoint( ), 716–717
getPoolSize( ), 908
getPort( ), 672, 683, 684
getPreciseWheelRotation( ), 718
getPreferredSize( ), 796–797
getPriority( ), 230, 240, 438
getProperties( ), 424, 519
getProperty( ), 424, 426, 520–521, 522
getPropertyDescriptors( ), 936, 937, 943
getQueuedTaskCount( ), 907
getRed( ), 756
getRegisteredParties( ), 882
getRemoveListenerMethod( ), 940
getRequestMethod( ), 679
getResponseCode( ), 679
getResponseMessage( ), 679
getRGB( ), 756
getRuntime( ), 417, 418
getScript( ), 539
getScrollAmount( ), 718
getScrollType( ), 718
getSelectedCheckbox( ), 782
getSelectedIndex( ), 784, 786, 982
getSelectedIndexes( ), 787
getSelectedItem( ), 784, 786, 985
getSelectedItems( ), 787
getSelectedText( ), 792, 795
getSelectedValue( ), 982–983
getServletConfig( ), 999
getServletContext( ), 1000
getServletInfo( ), 999
getServletName( ), 1000
ggetSession( ), 1005, 1009, 1010, 1015
ggetSize( ), 740, 754, 760
ggetSource( ), 710, 778, 975
ggetStackTrace( ), 222, 438, 446
ggetState( ), 256–257, 438, 780, 811
getStateChange( ), 715, 787
getStream( ), 705
getSuperclass( ), 431, 432
getSuppressed( ), 222, 306
getSurplusQueuedTaskCount( ), 907
getText( ), 775, 792, 795, 966, 967, 969
getTimeInstance( ), 928
getUnarrivedParties( ), 882
g getTotalSpace( ), 585
getUsableSpace( ), 585
getValue( ), 487, 489, 712, 789, 1008, 1010, 1014
getWheelRotation( ), 718
getWhen( ), 711
getWidth( ), 961
getWindow( ), 719
getWriter( ), 997, 1001
getX( ), 716
getXOnScreen( ), 717
getY( ), 716
getYOnScreen( ), 717
GIF image format, 829–830
Glass pane, 949
Glassfish, 994, 995
Glob, 655–656
Gosling, James, 6
goto keyword, 33
Goto statement
   using labeled break as form of, 104–106
   using labeled continue as form of, 107
grabPixels( ), 841–842
Graphics
   context, 307, 693, 749
   sizing, 754–755
Graphics class, 307, 308, 693, 694, 737, 749, 756, 764,
drawing methods, 749–753
GraphicsEnvironment class, 737, 760, 761
GregorianCalendar class, 531, 534–536, 539
Grid bag layouts, 805–810
GridBagConstraints class, 737, 806–808
constraint fields, table of, 806–807
GridBagLayout class, 737, 805–806, 808, 810
GridLayout class, 737, 801–802
gridwidth constraint field, 806, 808
group( ), 639, 874
GZIP file format, 579

H
Hash code, 471
Hash table, 470–471
hashCode( ), 185, 276, 399, 401, 404, 405, 406, 408, 414, 416, 427, 445, 446, 447, 458, 483, 487, 504, 516,
Hashing, 471

HashMap class, 487, 488–489, 490, 491, 516

HashSet class, 465, 470–472

Hashtable class, 466, 508, 516–519

    and iterators, 518

        legacy methods, table of, 517

hasMoreElements( ), 508, 526

hasMoreTokens( ), 526

hasNext( ), 477, 478

hasNextX( ) Scanner methods, 564, 566

    table of, 565

Headers, 677

HeadlessException, 740, 774

headMap( ), 484, 485

headSet( ), 461, 462

HEIGHT, 700
Hexadecimals, 41, 42
    as character values, 43
Hierarchical abstraction and classification, 18
    and inheritance, 19, 161
High surrogate char, 414
Histogram, 842
Hoare, C.A.R., 230
Holzmann, Gerard J., 840
HotSpot technology, 10
HSB (hue-saturation-brightness) color model, 756
HSBtoRGB( ), 756
HSPACE, 700, 701
HTML (Hypertext Markup Language), 993
    file for an applet, 308, 688, 699–701
HTTP, 668, 675, 993
    downloads, 1054
    GET requests, handling, 1010–1011
and HttpURLConnection class, 679
port, 668

POST requests, handling, 1010, 1012–1013
requests, 993, 994, 1004–1005, 1010
response, 993, 994, 997, 1004, 1005–1006
and URLConnection class, 677

HTTP session

stateful, 681

tracking, 1015–1016

HttpCookie class, 681

HttpServletRequest interface, 1004, 1008

methods, table of, 1009

HttpServletResponse interface, 1004, 1005

methods, table of, 1006

HttpSession interface, 1004, 1006, 1010, 1015
methods, table of several, 1007
HttpSessionBindingEvent class, 1004, 1010
HttpSessionBindingListener interface, 1004, 1006
HttpSessionEvent class, 1004, 1009, 1010
HttpURLConnection class, 679–681
hypot( ), 435

I
Icon interface, 966
Icons, Swing button, 969
Identifiers, 24, 32, 33, 44
IdentityHashMap class, 487, 491
IEEEremainder( ), 435
if statement, 28–29, 30, 81–84
  boolean variable used to control the, 82, 274
  nested, 83
  and recursive methods, 140
switch statement versus, 88–89
if-else-if ladder, 83–84

IllegalAccessException, 218, 221

IllegalArgumentOutOfRangeException, 220, 457, 459, 461, 463, 475, 483, 484, 485, 502, 503, 504

IllegalFormatException, 550

IllegalMonitorStateException, 220

IllegalStateException, 220, 457, 463, 465, 912, 1006

IllegalThreadStateException, 220

Image class, 737, 829, 830–831

ImageConsumer interface, 841–843, 844

ImageFilter class, 844–857

ImageIcon class, 965, 966

ImageObserver interface, 831, 832–833, 835

ImageProducer interface, 830, 839–840, 841, 844

imageUpdate( ), 832–833

bit flags, table of, 833

Images, 829–860
animation of, 857–860
creating, loading, displaying, 830–832
double buffering and, 833–836
stream model for, 844

Imaging, 829

IMG tag, 701

implements clause, 197

and generic interfaces, 348–349

import statement, 194–195

and static import, 319–321

in, 292, 293, 420, 423

Increment operator ( + + ), 30, 61, 64–66

indexOf( ), 382–383, 394, 395, 459, 460, 510, 511

IndexOutOfBoundsException, 220, 459

Inet4Address class, 671

Inet6Address class, 671

InetAddress class, 669–671, 683
InetSocketAddress class, 683

infinity (IEEE floating-point specification value), 402

inForkJoinPool( ), 907

INHERIT, 421

InheritableThreadLocal class, 444

Inheritance, 5, 19–21, 22–23, 142, 145, 161–186

and annotations, 288

and enumerations, 265

final and, 184–185

and interfaces, 187, 197, 206

multilevel, 171–174

and multiple superclasses, 163, 187

@Inherited built-in annotation, 286, 287

init( ), 689, 691, 692, 693, 695, 741, 994, 997, 999

and Swing, 957, 959

initCause( ), 222, 224

Inline method calls, 184
Inner classes, **149–152, 731–733**

  anonymous, **732–733**

InputEvent class, **710, 713–714, 715, 716**

InputMismatchException, **566**

InputStream class, **290, 291, 293, 563, 564, 590, 591, 592, 596, 599, 600, 602, 608, 628, 650**

  methods, table of, **591**

  objects, concatenating, **603–605**

InputStreamReader class, **292, 293**

insert( ), **392, 795**

Insets class, **737, 800–801, 961**

Instance of a class, **19, 109**

  See also Object(s)

Instance variables

  accessing, **111, 116, 117–118, 120**

  definition of, **19, 110**

  hiding, **125**
static, 145–146

as unique to their object, 111, 112–113

using super to access hidden, 170–171

instanceof operator, 310–312, 482

and generic classes, 355–357

InstantiationException, 221

Instrumentation interface, 450

int, 27, 35, 36, 37

and integer literals, 41

IntBuffer class, 631

Integer class, 269, 403, 410–411

constructors, 269

methods, table of, 406–407

Integer(s), 35, 36–38, 66–67

literals, 41–42

interface keyword, 187, 196

and annotations, 276
interfaceModifiers( ), 922

Interface(s), 187, 196–206

general form of, 196–197

generic. See Generic interfaces implementing, 197–199

and the inheritance hierarchy, 196

inheritance of, 205–206

member, 200

methods, 196, 197–198

nested, 200

reference variables, 198–199, 204

types for bounded types, using, 336

variables, 197, 204–205

Internet, 3, 6, 7, 8, 12, 16, 667

addresses, obtaining, 671

addressing scheme, 668

downloads, 1053–1054
and portability, 7, 8, 9

and security, 8–9

Internet Engineering Task Force (IETF) BCP 47, 539

Internet Protocol (IP)

definition of, 667

addresses, 668

InterNIC, 672, 674

InterruptedException, 221, 232, 842

Introspection, 934–937, 940, 943

Introspector class, 939, 940

intValue( ), 269, 398, 400, 401, 404, 405, 406, 408

InvalidPathException, 638

Investment, future value of an

applet for finding, 1028–1031

formula to compute, 1028

Investment required to achieve a future value

applet for finding, 1032–1036
formula to compute, 1032

invoke( ), 895, 896, 905

invokeAll( ), 895, 900, 907

invokeAndWait( ), 954, 959

invokeLater( ), 954, 959

I/O, 26, 289–307, 581–628

and applets, 307, 309

channel-based, 13, 290, 629. See also NIO; NIO and channel-based I/O

classes, list of, 581–582

console, 26, 93, 289, 293–297

error handling, 299–300

exceptions, 588–589

file, 297–306, 582–588

formatted. See I/O, formatted

interfaces, list of, 582

new. See NIO
redirection, 421

streams. See Streams I/O, formatted, 14, 548–573

format specifiers. See Format specifiers using Formatter, 548–562. See also Formatter class using printf( ), 155, 620

using Scanner, 563–573. See also Scanner class

io package. See java.io package

IOError, 620

IOException, 93, 293, 298, 301, 302, 588, 591, 592, 596, 602, 610, 613, 623, 624, 625, 635, 654, 657, 672, 682

ipadx constraint field, 806, 808

ipady constraint field, 806, 808

IPv4 (Internet Protocol, version 4), 668, 669, 670, 671

IPv6 (Internet Protocol, version 6), 668, 669, 670, 671

isAbsolute( ), 584, 635
isAlive( ), 230, 238–240, 438

isAltDown( ), 714

isAltGraphDown( ), 714

isAnnotationPresent( ), 282, 284, 445

isBound( ), 672, 684, 940

isCancelled( ), 906

isClosed( ), 672

isCompletedAbnormally( ), 906

isCompletedNormally( ), 906

isConnected( ), 672, 684

isConstrained( ), 940

isControlDown( ), 714

isDigit( ), 412, 413, 414

isDirectory( ), 585, 636, 638

isEditable( ), 793, 795

isEmpty( ), 388, 457, 458, 483, 510, 515, 517, 527

isEnabled( ), 811
isExecutable( ), 636, 652
isFile( ), 584
isHidden( ), 585, 636, 639
isInfinite( ), 400, 401, 402–403
isLeapYear( ), 535
isLetter( ), 412, 413, 414
isLowercase( ), 412, 413
isMetaDown( ), 714
isMulticastAddress( ), 671
isMultipleMode( ), 822
isNaN( ), 400, 401, 402–403
ISO-Latin-1 character set, 39, 43
isPopupTrigger( ), 717
isPublic( ), 921
isQuiescent( ), 907
isReadable( ), 636, 652
isSelected( ), 972, 973, 974, 975
isSet array, 531
isSet( ), 532
isShiftDown( ), 714
isShutdown( ), 908
isTemporary( ), 713
isTerminated( ), 908
isTimeSet, 531
isUppercase( ), 412, 413
isWhitespace( ), 412 413
isWritable( ), 636, 639, 652
ItemEvent class, 710, 714–715, 780, 784, 787, 812, 972, 973
ItemListener interface, 720, 721, 780, 784, 812, 972, 973
ItemSelectable interface, 715
itemStateChanged( ), 721, 780, 784, 972, 973
Iterable interface, 449, 456, 457, 479, 482, 509
Iterable <Path> interface, 634, 654

Iteration statements, 81, 89–102

Iterator, 455, 456, 476–480
  and maps, 482
  and PriorityQueue, 474
  and synchronized collections, 499

Iterator interface, 455, 457, 476–477, 478–479, 507, 508
  methods, table of, 477

iterator(), 449, 458, 459, 477–478, 654

J

J2SE 5, new features of, 13–14

JApplet class, 687, 949, 957, 959

Java

  API packages, table of core, 909–911
  and C, 3, 5, 7, 11
  and C++, 3, 7, 11
and C#, 8

design features (buzzwords), 10–13

history of, 3, 6–8, 13–16

and the Internet, 3, 6, 7–9, 12, 16, 667, 1053

as interpreted language, 9, 10, 12

keywords, 33–34

as strongly typed language, 10, 11, 35, 41

versions of, 13–14

and the World Wide Web, 6, 7, 11

Java Archive (JAR) files, 578

Java Beans, 432, 451, 909, 919, 933–943

advantages of, 934

API, 938–940

customizers, 937

demonstration program, 940–943

introspection, 934–937, 940, 943

properties. See Property, Java Bean
serialization, 937
.java filename extension, 23
Java Community Process (JCP), 16
Java EE SDK, 994, 998
Java Foundation Classes (JFC), 946
java (Java application launcher). See Application launcher (java)
Java Native Interface (JNI), 313
Java Network Launch Protocol (JNLP), 688, 699
java package, 194
Java SE 7, 14–16
Java Virtual Machine (JVM), 9–10, 12, 13, 16, 24, 25, 417, 437, 451
java.applet package, 289, 307, 687
java.awt package, 707, 710, 736, 829, 830, 956
classes, tables of some, 736–737
java.awt.Dimension class, 919
java.awt.event package, 707, 709, 710, 720, 729, 954, 956

event classes, table of commonly used, 710

interfaces, table of commonly used, 720

java.awt.image package, 829, 839, 844, 860

java.beans package, 936, 938–940

classes, table of, 938–939

interfaces, tables of, 938

java.io package, 289, 290–292, 298, 304, 581–582, 588, 629, 652

classes, list of, 581–582

interfaces, list of, 582

java.io.Externalizable interface, 937

java.io.IOException, 93

java.io.Serializable interface, 937

java.lang package, 194, 220–221, 277, 286, 292, 298, 304, 348, 371, 397–451, 588

classes and interfaces, list of, 397
implicit importation of the package, java.lang.annotation, java.lang.annotation.RententionPolicy enumeration, java.lang.image, java.lang.instrument, java.lang.invoke, java.lang.management, java.lang.ref, java.lang.reflect classes, table of, java.net classes and interfaces, list of, java.nio, java.nio.channels, java.nio.channels.spi, java.nio.charset
java.nio.charset.spi package, 629
java.nio.file package, 629, 634
java.nio.file.attribute package, 629, 634, 638
java.nio.file.spi package, 629, 634
java.rmi package, 909, 924
java.text package, 909, 927
java.util package, 453–454, 508, 525, 707, 709
  classes, list of top-level, 453–454
  interfaces defined by, list of, 454
java.util.concurrent package, 578, 862–863, 888
java.util.concurrent.atomic package, 578, 862, 863, 892
java.util.concurrent.locks package, 578, 862, 863, 889, 890, 892
java.util.jar package, 578
java.util.logging package, 578
java.util.prefs package, 578
java.util.regex package, 579, 909, 911
java.util.spi package, 579
java.util.zip package, 579
javac (Java compiler), 23–24, 188
    and generics, 325
javadoc
tags, 1079–1083
    utility program, 1079, 1083
javah.exe, 314, 315
javap, 364
javax.imageio package, 860
javax.servlet package, 997, 998–1002
    interfaces and classes, list of core, 998–999
javax.servlet.http package, 998, 1004–1010
    interfaces and classes, list of core, 1004
javax.swing package, 948, 950, 951, 965, 987
javax.swing.event package, 954, 967, 982, 987
javax.swing.table package, 990
javax.swing.tree package, 987

JButton class, 949, 956, 965, 969–971

JCheckBox class, 965, 969, 971, 973–975

 JComboBox class, 965, 984–986

 JComponent class, 948, 949, 957, 960, 961, 965

 JDialog class, 949

 JDK 7 (Java SE 7 Development Kit), 14, 23

 JFrame class, 949, 950, 951–952, 953, 963

 JIT (Just-In-Time) compiler, 10, 12

 JLabel class, 950, 952, 954, 965–967

 JLayeredPane class, 949

 JList class, 965, 981–984

 jni.h, 315

 jni_md.h, 315

 join( ), 230, 238–240, 439, 894, 895, 903, 905

 Joy, Bill, 6
 JPanel class, 949, 963, 977
 JPEG image format, 830
 JRadioButton class, 965, 969, 971, 975–977
 JRootPane class, 949
 JScrollBar class, 949
 JScrollPane class, 965, 979–981, 982, 987, 988, 990
 JTabbedPane class, 965, 977–979
 JTable class, 965, 990–992
 JTextComponent class, 967
 JTextField class, 965, 967–968, 1025
 JToggleButton class, 965, 969, 971–973, 975
 JToggleButton.ToggleButtonModel class, 971
 JTree class, 965, 986–989
 Jump statements, 81, 102–108
 Just In Time (JIT) compiler, 10, 12
 JVM (Java Virtual Machine), 9–10, 12, 13, 16, 24, 25, 417, 437, 451
JWindow class, 949

K

Kernighan, Brian, 4

Key codes, virtual, 715–716, 728

KeyAdapter class, 730

Keyboard events, handling, 726–729

KeyEvent class, 710, 712, 713, 715–716

KeyListener interface, 720, 722, 726–729, 730

keyPressed(), 722, 726, 727

keyReleased(), 722, 726

keys(), 515, 517

keySet(), 483, 484, 518, 575, 681

keyTyped(), 722, 726, 727

Keywords, table of Java, 33

L

Label

AWT standard control, 775–776
Swing, 950, 952, 965–967
used with break statement, 104
used with continue statement, 107
Label class, 737, 775–776
last( ), 461, 803
lastElement( ), 510, 511
lastIndexOf( ), 382–383, 395, 459, 460, 510, 511
lastKey( ), 484
Late binding, 184
Layered pane, 949
Layout managers, 739, 773, 796–810
default, 773, 796, 797
LayoutManager interface, 796
length instance variable of arrays, 147–149
length( ), 153, 374, 389–390, 448, 527
Lexer (lexical analyzer), 525
Libraries, class, 23, 34
Lindholm, Tim, 6

LineNumberInputStream deprecated class, 582

LineNumberReader class, 292

LinkedBlockingDeque class, 889

LinkedBlockingQueue class, 889

LinkedHashMap class, 487, 490–491

LinkedHashSet class, 465, 472

LinkedList class, 465, 469–470

  example program using the, 480–481

LinkedTransferQueue, 889

List class, 737, 786

  extending, 826–827

List controls, 720, 786–788

List interface, 456, 459, 466, 469, 478, 501, 509, 511

  methods, table of, 460

List, Swing, 981–984

list( ) and directories, 583, 585–587
listFiles( ), 587

ListIterator interface, 457, 476–477, 478–479, 507
  methods, table of, 477

listIterator( ), 460, 478

ListModel, 982

ListResourceBundle class, 575

ListSelectionEvent class, 982, 983, 990

ListSelectionListener interface, 982, 983

ListSelectionModel, 982, 990

Literals, 32, 41–44
  class, 279
  regular expression, 913
  string, 374

load( ), 418, 424, 520, 522–524

loadLibrary( ), 314, 418, 424

Loan balance, applet to find, 1044–1047

Loan payments
applet to compute, 1020–1028
formula for calculating, 1020

servlet to compute, 1048–1051

Locale class, 387, 538–539, 927, 928

Locale Data Markup Language (LDML), 539

Locale.Builder class, 539

Lock interface, 863, 890
    methods, table of, 890

lock( ), 863, 890

lockInterruptibly( ), 890

Locks, 889–892

log( )
    math method, 434
    servlet method, 1000, 1002

log10( ), 434

log1p( ), 434

Logical operators
bitwise, 67–69

Boolean, 75–77

long, 35, 36, 37–38

literal, 41

Long class, 269, 403, 410

methods, table of, 408–409

LongBuffer class, 631

longValue( ), 269, 398, 400, 401, 404, 405, 406, 408

Look and feels, 946–947

lookup( ), 925

loop( ), 706

Loop(s), 81

Boolean object to control, using a, 274

continue statement and, 106–107

do-while, 90–93

for. See for loop

infinite, 96–97, 103
nested, 102, 104, 105–106

with polling, event, 228, 245

while, 89–90

Low surrogate char, 414

M

main( ), 25–26, 110, 142, 145

and applets, 26, 110, 308, 309, 688

and the java application launcher, 25

and command-line arguments, 25, 154–155

and Swing programs, 953–954

and windowed applications, 747, 748

main (default name of main thread), 232

MalformedURLException, 675

Map interface, 482–484, 486, 488, 491, 515, 516, 518

methods, table of, 483

map( ), 633, 644, 647, 662, 664

Map(s), 455, 482–492
classes, 487–492

collection-view of a, obtaining a, 455, 482, 483–484

interfaces, 482–487

Map.Entry interface, 482, 486–487

methods, table of, 487

MapMode.PRIVATE, 644

MapMode.READ_ONLY, 644

MapMode.READ_WRITE, 644, 647

MappedByteBuffer class, 631, 644

mark( ), 591, 592, 597, 600, 603, 611, 616, 631

markSupported( ), 591, 600, 603, 610, 611, 616

Matcher class, 911, 912

matcher( ), 912

matches( ), 388, 912, 914, 919

Math class, 45, 131, 433–436

rounding methods, table of, 434–435

and static import example, 319–321
max( ), 434, 497, 500

MAX_EXPONENT, 399

MAX_PRIORITY, 240, 437

MAX_RADIX, 411

MAX_VALUE, 399, 403, 411

MediaTracker class, 737, 829, 836–839

Member, class, 19, 110

access and inheritance, 163–164

access, table of, 191

controlling access to, 141–144

static, 145–146

Member interface, 451, 919

Memory

allocation using new, 51–52, 53, 113–114

leaks, 298, 304

management, in Java, 11–12, 125

and Runtime class, 418–419
MemoryImageSource class, 839–840, 844

Menu bars and menus, 773, 810–816

  action command string of, 812

  and events, 812

Menu class, 737, 810, 811

Menu item as an event source, 720

MenuBar class, 737, 810, 811

MenuItem class, 737, 810–811, 812

Metadata, 276

  See also Annotation(s)

Method class, 278, 281, 282, 451, 920, 921, 940

Method(s), 19, 110, 115–121

  abstract, 181–184

  and annotations, 276, 288

  and autoboxing, 271–272

  bridge, 362–364

  calling, 117, 119
dispatch, dynamic, 178–181

and the dot (.) operator, 111, 117, 118

factory, 669

final, 147, 184

general form, 116

generic, 326, 338, 344–346, 366

getter, 934

hidden, using super to access, 170–171, 176

inlining, 184

interface, 196, 197–198

lookup, dynamic, 198

native, 312–316

overloading, 129–134, 158–160, 177

overriding. See Overriding, method

and parameters, 116, 119–121

passing an object to, 137–138

recursive, 139–141
resolution, dynamic, 196
returning an object from, 138–139
returning a value from, 118–119
scope defined by, 45–47
setter, 934
static, 145–146
subclasser responsibility, 182
synchronized, 230, 241–243
type inference and, 360
varargs. See Varargs
variable-arity, 155
MethodDescriptor class, 936, 939, 940
MethodHandle class, 451
methodModifiers( ), 922
MethodType class, 451
MIME (Multipurpose Internet Mail Extensions), 993, 997
min( ), 434, 497, 500
minimumLayoutSize( ), 796
MIN_EXPONENT, 399
MIN_NORMAL, 399
MIN_PRIORITY, 240, 437
MIN_RADIX, 411
MIN_VALUE, 399, 403, 411
mkdir( ), 588
mkdirs( ), 588
Model-Delegate component architecture, 947–948
Model-View-Controller (MVC) component architecture, 947
Modifier class, 921, 922
“is” methods, table of, 923
Modulus operator (%), 61, 63
Monitor, 230, 241, 243
Mouse events, handling, 723–726
MouseAdapter class, 730
mouseClicked( ), 722, 730
mouseDragged( ), 722, 729
mouseEntered( ), 722
MouseEvent class, 710, 712, 713, 716–717
mouseExited( ), 722
MouseListener interface, 720, 722, 723–726, 730
MouseMotionAdapter class, 729, 730
MouseMotionListener interface, 709, 720, 722, 723–726, 729, 730
mouseMoved( ), 722, 729
mousePressed( ), 722
mouseReleased( ), 722
MouseWheelEvent class, 710, 717–718
MouseWheelListener interface, 720, 722, 723
mouseWheelMoved( ), 722
Multicore systems, 228–229, 257, 862, 863, 893–894,
Multitasking, 227, 229

Multithreaded programming, 7, 11, 12, 227–257

context switching. See Context switching effectively using, 257

and multicore versus single-core systems, 228

Observable class, Observer interface and, 544

and spurious wakeup, 245

and StringBuilder class, 395

and synchronization. See Synchronization and threads. See Thread(s)

versus the concurrency utilities, traditional, 861, 908

and parallel programming, 893–894

versus single-threaded system, 228

MutableComboBoxModel, 985

MutableTreeNode interface, 987

MVC (Model-View-Controller) component architecture,
Name-space collisions
   between instance variables and local variables, 125
   packages and, 187, 188, 321
Naming class, 924, 925
NaN, 399, 402
nanoTime(), 424, 425, 900
native modifier, 313
Natural ordering, 448, 492
Naughton, Patrick, 6
NavigableMap interface, 482, 484–486, 489
   methods, table of, 485–486
NavigableSet interface, 456, 461, 472, 473
   methods, table of, 462
Negative numbers in Java, representation of, 66, 67
NEGATIVE_INFINITY, **399**

NegativeArraySizeException, **220, 501**

.NET Framework, **8**

NetBeans, **994, 995**

Networking, **667–685**

  - basics, **667–668**
  - classes and interfaces, list of, **668–669**
  - new, **51–52, 53, 113–114, 121, 123, 125, 139, 182**
  - autoboxing and, **271**
  - and enumerations, **260, 263**
  - and type inference, **359–360**

NEW, **256**

New I/O. See NIO

newByteChannel( ), **633, 636, 641, 644, 645, 646**

newCachedThreadPool( ), **883**

newCondition( ), **890**

newDirectoryStream( ), **636, 654, 655–657**
newFixedThreadPool( ), 883
newInputStream( ), 637, 649, 650–651
newOutputStream( ), 637, 649, 651
newScheduledThreadPool( ), 883
next( ), 477, 478, 566, 803
nextAfter( ), 435
nextBoolean( ), 539, 566
nextBytes( ), 539
nextDouble( ), 205, 539, 540, 566, 568, 570
nextElement( ), 508–509, 526, 605
nextFloat( ), 539, 540, 566
nextGaussian( ), 539, 540
nextInt( ), 539, 566, 570
nextLong( ), 539, 566
nextToken( ), 526
nextUp( ), 435
nextX( ) Scanner methods, 564, 566, 568, 570
NIO, 15, 581, 629–665

packages, list of, 629

pre-JDK 7 NIO versus new, 660

reading a file using pre-JDK 7, 660–663

for path and file system operations, using, 652–659

for stream-based I/O, using, 649–651

writing to a file using pre-JDK 7, 663–665

NIO and channel-based I/O

copying a file using, 648–649

reading a file using, 641–645

writing to a file using, 645–648

NIO.2, 629, 640, 652

NORM_PRIORITY, 240, 437

NoSuchElementException, 461, 463, 484, 509, 566, 573

NoSuchFieldException, 221
NoSuchMethodException, 221, 278

NOT operator

    bitwise unary (~), 66, 67, 68–69

    Boolean logical unary (!), 75–76

NotDirectoryException, 654

notepad, 420, 423

notify( ), 185, 186, 245, 247–249, 254–255, 427, 890, 908

notifyAll( ), 185, 186, 245, 427

notifyObservers( ), 541, 542

NotSerializableException, 627

null, 34, 113

Null statement, 90

NullPointerException, 217, 220, 457, 459, 461, 463, 475, 483, 484, 485, 501, 516, 573, 605

Number class, 269, 398

NumberFormat class, 927, 1024
NumberFormatException, 221, 269, 702

Numbers, formatting, 551–552, 555–557, 1024

O

Oak, 6

Object class, 185–186, 326, 427
    as a data type, problems with using the, 330–332, 506
    methods, table of, 185, 427

Object reference variables
    and abstract classes, 182, 184
    assigning, 115
    declaring, 113
    and cloning, 427–428
    and dynamic method dispatch, 178–181
    interface, 198–199
    to superclass reference variable, assigning subclass, 166, 170
OBJECT tag, 688, 701

Object-oriented programming (OOP), 5, 6, 17–23, 109
  model in Java, 11

Object(s), 19, 109, 114
  bitwise copy (clone) of, 427
  creating, 111, 113–114
  initialization with a constructor, 121, 123–124
  to a method, passing, 137–138
  monitor, implicit, 230, 243
  as parameters, 134–136
  returning, 138–139

  serialization of. See Serialization type at run time, determining, 310–312

Object.notify( ), 890

Object.wait( ), 890

ObjectInput interface, 625
  methods defined by, table of, 625
ObjectInputStream class, 291, 625
    methods defined by, table of, 626
ObjectOutput interface, 623
    methods defined by, table of, 623
ObjectOutputStream class, 291, 624
    methods defined by, table of, 624
Objects class, 577
Observable class, 541–544
    methods, table of, 541
Observer interface, 541–544
Octals, 41
    as character values, 43
of( ), 475, 476
offer( ), 463, 474
offerFirst( ), 463, 464, 469
offerLast( ), 463, 464, 469
offsetByCodePoints( ), 388, 395
onAdvance( ), 879, 882
open( ), 633
openConnection( ), 676, 678, 679
OpenOption interface, 635
Operator(s)

  arithmetic, 61–66

  assignment. See Assignment operator(s)

  bitwise, 66–74

Boolean logical, 75–77
conditional-and, 77
conditional-or, 77
diamond (<>), 359–360
parentheses and, 79
precedence, table of, 78
relational, 28, 41, 74–75
ternary if-then-else (?:), 77–78
OR operator (|)
bitwise, 66, 67, 68–69

Boolean logical, 75–76

OR operator, short-circuit (||) Boolean logical, 75, 76–77

Oracle, 14

Ordinal value enumeration constants, 265–266

ordinal( ), 265, 266, 447

out output stream, 26, 292, 293, 420, 423

out( ), 549, 551

OutputStream class, 290, 291, 296, 590, 591, 599, 601, 605, 607, 619, 624, 628, 651

methods, table of, 592

OutputStreamWriter class, 292

Overloading methods, 129–134, 158–160, 177

@Override, built-in annotation, 286, 287

Overriding, method, 175–181

and dynamic method dispatch, 178–181
final to prevent, using, 184

in a generic class, 358–359

and run-time polymorphism, 178, 179, 181

P

Package(s), 142, 187–196, 206

access to classes contained in, 190–194, 195

core Java API, table of, 909–911

defining, 188

finding, 188–189

importing, 194–196

Swing, 950

version data, obtaining, 444

Package class, 282, 444–445

methods, table of, 444–445

package statement, 188, 194

Paint mode, setting, 757–759

paint( ), 297, 691, 692, 693, 694, 695, 696, 698, 742,
lightweight components and overriding, 828

Paintable area, computing, 961

paintBorder( ), 960

paintChildren( ), 960

paintComponent( ), 960, 963, 964

Painting in Swing, 959–964

Panel class, 688, 737, 738, 739, 803

Panes, Swing container, 949. See also Content pane

Parallel programming. See Programming, parallel

PARAM NAME, 700, 701

Parameter(s), 25, 116, 119–121

applets and, 701–703

and constructors, 123–124

objects as, 134–136

and overloaded constructors, 134

and overloaded methods, 129, 177
and the scope of a method, \textit{46}

servlet, reading, \textit{298–300}

type. See Type parameter(s)

variable-length (varargs), \textit{157}

Parameterized types, \textit{326, 328}

\texttt{parseBoolean( )}, \textit{416}

\texttt{parseByte( )}, \textit{404, 410}

\texttt{parseDouble( )}, \textit{401}

\texttt{parseFloat( )}, \textit{400}

\texttt{parseInt( )}, \textit{407, 410}

\texttt{parseLong( )}, \textit{409, 410}

\texttt{parseShort( )}, \textit{405, 410}

Parsing, definition of, \textit{525}

Pascal, \textit{4}

Passwords, reading, \textit{620}

Path interface, \textit{582, 585, 634–635, 640, 641, 652, 654, 660}
instance for stream-based I/O, using a, 649–651
methods, table of a sampling of, 634–635
obtaining an instance of the, 638, 640

Paths class, 638, 640

Pattern class, 911–912

Pattern matching. See Regular expressions

PatternSyntaxException, 913

Payne, Jonathan, 6

peek( ), 463, 513

peekFirst( ), 464, 469

peekLast( ), 464, 469

Peers, native, 945–946, 828

Persistence (Java Beans), 937

Phaser class, 862, 875–882

compatability with fork/join, 908

PI (Math constant), 433

PIPE, 421
PipedInputStream class, 291
PipedOutputStream class, 291
PipedReader class, 292
PipedWriter class, 292
PixelGrabber class, 841–843, 844
play( ), 690, 706
Pluggable look and feel (PLAF), 946–947, 948, 992
PNG file format, 830, 831
Point class, 717, 747
Pointers, 59, 113
poll( ), 463, 474
pollFirst( ), 462, 464, 469
Polling, 228, 245
pollLast( ), 462, 464, 469
Polygon class, 737, 753
Polymorphism, 5, 21–23
and dynamic method dispatch, 178–181, 182
and interfaces, 196, 199–200, 204
and overloaded methods, 129, 131, 132

pop( ), 464, 465, 513

PopupMenu class, 737, 816

Port, 667, 675

Portability problem, 6–7, 8, 9, 10, 12, 16
and data types, 36
and native methods, 316
and thread context switching, 229

POSITIVE_INFINITY, 399

PosixFileAttributes class, 639, 654

PosixFileAttributeView interface, 639

postVisitDirectory( ), 658

pow( ), 319–321, 434

preferredLayoutSize( ), 796

previous( ), 477, 803

preVisitDirectory( ), 658
print( ), 27, 34, 296, 297, 376, 606, 620, 1002

printf( )

    function, C/C++, 548

    method, Java, 155, 562, 606–607, 619, 620, 621

println( ), 26, 27, 34, 186, 296, 297, 376, 606, 619, 620, 1002

    and applets, 706

    and String objects, 58

printStackTrace( ), 222

PrintStream class, 291, 293, 296, 562, 605–607

PrintWriter class, 292, 296–297, 562, 619–620, 997

PriorityBlockingQueue class, 889

PriorityQueue class, 465, 473–474

private access modifier, 25, 142–144, 190–191

    and inheritance, 163–164

Process class, 416–417, 420, 421

    methods, table of, 417
Process, definition of, 416
Process-based versus thread-based multitasking, 227
processActionEvent( ), 823, 827
processAdjustmentEvent( ), 823, 827
ProcessBuilder class, 416, 421–423
   methods, table of, 421–422
ProcessBuilder.Redirect class, 421
ProcessBuilder.Redirect.Type enumeration, 421
processComponentEvent( ), 823
processFocusEvent( ), 823
processItemEvent( ), 823, 824, 826
processKeyEvent( ), 823
processMouseEvent( ), 823
processMouseMotionEvent( ), 823
processMouseWheelEvent( ), 823
processTextEvent( ), 823
Programming
multithreaded. See Multithreaded programming
object-oriented. See Object-oriented programming
process-oriented, 17, 18, 22
structured, 4, 5
Programming, parallel, 15, 229, 862, 863, 893–894
and specifying the level of parallelism, 896, 900–903
Project Coin, 15
Properties class, 454, 508, 519–522
methods, table of, 520
Properties, environment, 426
Property, Java Bean, 940
bound and constrained, 937
design patterns for, 934–935, 937, 940
PropertyChangeEvent, 937
PropertyChangeListener interface, 937, 938
PropertyDescriptor class, 936, 939, 940, 942
PropertyPermission class, 577

PropertyResourceBundle class, 575

PropertyVetoException, 937

protected access modifier, 126, 142, 190–191

public access modifier, 25, 142–143, 190–191

Push buttons, 691, 776–779
  action command string of, 776, 778, 779, 969
  Swing, 954–957, 969–971

push( ), 464, 465, 513

Pushback, 602

PushbackInputStream, 291, 599, 602–603

PushbackReader class, 292, 618–619

put( ), 483, 488, 489, 491, 515, 517
  and buffers, 631, 632, 646–647, 664

putAll( ), 483, 491

PutField inner class, 624
Query string, 1011

Queue interface, 456, 462–463, 469, 473, 474
  methods, table of, 463

quietlyInvoke( ), 907

quietlyJoin( ), 907

R

Race condition, 243

Radio buttons, 782
  Swing, 975–977

Radix, 403

radix( ), 573

Random class, 205, 539–540
  methods, table of, 539

random( ), 435

RandomAccess interface, 457, 482

RandomAccessFile class, 609–610, 632, 664

range( ), 475, 476
Raw types, 349–352, 508

and erasure, 362

READ, 421

read( ), 93, 292, 293–294, 298–299, 303, 449, 591, 600, 602, 611, 618, 625, 626, 633, 641, 650, 661

and end-of-file condition, 303

Readable interface, 449, 563, 569, 610

ReadableByteChannel interface, 563

readAttributes( ), 637, 639, 652

readBoolean( ), 608, 626

readDouble( ), 608, 626

Reader class, 291, 292, 293, 590, 610, 612, 614, 628

methods defined by, table of, 611

readExternal( ), 623

readInt( ), 608, 626

readLine( ), 294–295, 410, 620, 621, 1002

readObject( ), 625, 626
readPassword( ), 620, 621

ReadWriteLock interface, 892

Real numbers, 38

rebind( ), 924

receive( ), 683

Recursion, 139–141

and the Fork/Join Framework divide-and-conquer strategy, 897

RecursiveAction class, 863, 894, 895, 897, 898, 899, 903

RecursiveTask class, 863, 894, 895–896, 897

example program using, 903–905

Redirect class, 421

ReentrantLock, 890

ReentrantReadWriteLock, 892

Reflection, 277, 451, 909, 919–923

and annotations, 277–282
ReflectiveOperationException, 221

regionMatches( ), 379–380

register( ), 876

Regular expressions, 389, 564, 571, 909, 911–919
  syntax, 913
  wildcards and quantifiers, 911, 913, 915–916

reinitialize( ), 906

Relational operators, 28, 41, 74–75

release( ), 864–867

Remote interface, 924

Remote method invocation (RMI), 12, 622, 909, 923–927

RemoteException, 924

remove( ), 457, 458, 460, 463, 471, 477, 483, 515, 516, 517, 774, 953

removeActionListener( ), 956

removeAll( ), 457, 458, 774
removeAttribute( ), 1007, 1015
removeEldestEntry( ), 491
removeElement( ), 510, 511
removeElementAt( ), 510, 511
removeFirst( ), 464, 469
removeKeyListener( ), 709
removeLast( ), 465, 469
removeTListener( ), 936
removeTypeListener( ), 709
renameTo( ), 584
repaint( ), 696–698, 742, 960
replace( ), 385, 393–394
replaceAll( ), 388, 498, 912, 917–918
replaceFirst( ), 388
replaceRange( ), 795
ReplicateScaleFilter class, 844
reset( ), 573, 591, 592, 597, 599, 600, 603, 611, 616,
resolve( ), 634, 635

Resource bundles, 573–577

ResourceBundle class, 573–575
  methods, table of, 574–575

ResourceBundle.Control class, 573

resume( ), 13, 251–254, 437, 443

retainAll( ), 457, 458

@Retention built-in annotation, 277, 286

RetentionPolicy enumeration, 276–277, 450

return statement, 108, 116

reverse( ), 392–393, 407, 409, 498

reverseBytes( ), 405, 407, 409

reverseOrder( ), 498, 500

rewind( ), 631, 643, 646, 647, 661, 664

RGB (red-green-blue) color model, 756
  default, 840
RGBImageFilter class, 844, 845–857

RGBtoHSB( ), 756

Richards, Martin, 4

rint( ), 435

Ritchie, Dennis, 4

RMI compiler (rmic), 925–926

rmi protocol, 925

rmiregistry (RMI registry), 926

round( ), 435

Run-time

system, Java, 9. See also Java Virtual Machine (JVM)
type information, 13, 310, 355, 357, 432

run( ), 230, 233, 234, 437, 439, 544, 545, 907, 954

overriding, 235, 236, 544

using a flag variable with, 255–256

Runnable interface, 230–231, 437, 544, 905, 907, 954

implementing the, 233–235, 236
Runtime class, 416, 417–420, 903

executing other programs and, 420

memory management and, 418–419

methods, table of some, 417–418

RUNTIME retention policy, 276–277, 278, 281

RuntimeException class, 208–209, 217, 220, 224

RuntimePermission class, 445

S

@SafeVarargs built-in annotation, 286, 287

save( ), 519

scalb( ), 434

Scanner, 525

Scanner class, 563–573

closing an instance of the, 569

constructors, 563–564

delimiters, 564, 570–572

demonstration programs, 567–570
hasNextX() methods, table of, 565

how to use, 564, 566

methods, miscellaneous, 572–573

nextX() methods, table of, 566

schedule( ), 545–546

ScheduledExecutorService interface, 883

ScheduledThreadPoolExecutor class, 863, 883

Scientific notation, 42, 551–552

Scopes in Java, 45–47

Scroll bars, 720, 788–791, 979

Scroll pane, 979–981

Scrollbar class, 737, 789

    extending, 827–828

search( ), 513–514

Security manager, 298, 994

Security problem, 8, 9–10, 16

    and native methods, 316
and servlets, 994

SecurityException, 220, 298, 417, 423, 589, 606, 654, 657

SecurityManager class, 445

seek( ), 610

SeekableByteChannel interface, 633, 641, 644, 645

select( ), 784, 787, 792, 795

Selection statements, 81–89

Selectors, 633

Semaphore, 861, 862, 863–869

and setting initial synchronization state, 869

Semaphore class, 862, 863–864

send( ), 683

Separable Model architecture, 947

Separators, 33

SequenceInputStream class, 291, 603–605

Serializable interface, 622–623, 907
Serialization, 622–628

example program, 626–628

and Java Beans, 937

and static variables, 623

and transient variables, 623, 627

Server, 667

ServerSocket class, 632, 671, 681–682

service( ), 994, 997, 999

ServiceLoader class, 577

Servlet interface, 998, 999

methods, table of, 999

Servlet(s), 10, 16, 993–1016, 1019

advantages of, 994

API, 998

development options, 994–1000

example program for a simple, 996–998

financial calculation example, 1048–1052
life cycle of, 994
parameters, reading, 1002–1004
and portability, 10
and security, 994
and session tracking, 1015–1016
using Tomcat to develop, 994, 995–996
ServletConfig interface, 998, 1000
ServletContext interface, 998, 1000
methods, table of various, 1000
ServletException class, 999, 1002
ServletInputStream class, 999, 1002
ServletOutputStream class, 999, 1002
ServletRequest interface, 997, 998, 1000, 1002
methods, table of various, 1001
ServletResponse interface, 997, 998, 1000
methods, table of various, 1001
Session tracking, HTTP, 1015–1016
Set interface, 456, 459–460, 470, 475, 484, 486
Set-view, obtaining, 483, 488–489, 518
set( ), 459, 460, 470, 477, 528, 532–533, 892
setActionCommand( ), 779, 812, 969, 975
setAddress( ), 684
setAlignment( ), 775
setAttribute( ), 1000, 1007, 1015
setBackgroundColor( ), 694, 755
setBlockIncrement( ), 790
setBorder( ), 963–964
setBounds( ), 739, 796
setChanged( ), 541
setCharAt( ), 390–391
setColor( ), 756
setConstraints( ), 806
setContentTypes( ), 997, 1001
setData( ), 684
setDefault( ), 536, 538
setDefaultCloseOperation( ), 952
setDisabledIcon( ), 969
setEchoChar( ), 793
setEditable( ), 793, 795, 1025
setEnabled( ), 811
setFollowRedirects( ), 679
setFont( ), 762
setForeground( ), 694, 755
setIcon( ), 966
SetIntField( ), 315
setLabel( ), 776, 780, 811
setLastModified( ), 585
setLayout( ), 796, 953
setLength( ), 390, 610, 684
setLocation( ), 739
setMaxAge( ), 1008, 1015
setMultipleMode( ), 822
setName( ), 232, 439
setPaintMode( ), 758
setPort( ), 684
setPreferredSize( ), 739, 790
setPressedIcon( ), 969
setPriority( ), 240, 439
setReadOnly( ), 585
setRequestMethod( ), 680
setRolloverIcon( ), 969
setSelectedCheckbox( ), 782
setSelectedIcon( ), 969
setSelectionMode( ), 982
setSize( ), 510, 739, 740, 741, 952
setSoTimeout( ), 683
setStackTrace( ), 222
setState( ), 780, 811
setStream( ), 705
setText( ), 775, 792, 795, 966, 969, 1025
setTitle( ), 740
setUnitIncrement( ), 790
setValue( ), 487, 789, 1008
setValues( ), 789
setVisible( ), 740, 741, 953
setXORMode( ), 757, 758–759
Sheridan, Mike, 6
Shift operators, bitwise, 66, 69–73
Short class, 269, 403, 410
  methods defined by, table of, 405
short data type, 35, 36, 37, 41
ShortBuffer class, 631
shortValue( ), 269, 398, 400, 402, 404, 405, 407, 409
show( ), 803
showDocument( ), 704–706
showStatus( ), 690, 699, 706
shuffle( ), 498, 500
shutdown( ), 882, 885, 897, 908
shutdownNow( ), 908
Sign extension, 71
signal( ), 890
signum( ), 407, 409, 435
SimpleBeanInfo class, 937, 943
SimpleDateFormat class, 539, 929–930
  formatting string symbols, table of, 929
SimpleFileVisitor class, 658, 659
SimpleTimeZone class, 537–538
sin( ), 38, 433
SingleSelectionModel, 977
sinh( ), 433
SIZE, 399, 403
size( ), 457, 458, 471, 483, 511, 515–516, 517, 528,
skip( ), 573, 591, 592–594, 600, 611, 625

sleep( ), 230, 232, 237, 238, 439, 888

slice( ), 631

Slider box, 789

Socket class, 632, 671–675, 680, 682

Socket(s)

  overview, 667

  TCP/IP client, 671–675

  TCP/IP server, 671, 681–682

SocketAddress class, 683

SocketChannel class, 632, 633

SocketException, 683

sort( ), 498, 503–504

SortedMap interface, 482, 484

  methods, table of, 484

SortedSet interface, 456, 460–461
methods, table of, 461
Source code file, naming a, 23–24
SOURCE retention policy, 276–277
split( ), 388, 918
sqrt( ), 38, 45, 319–321, 434
Stack
definition of, 21, 126–127
ways to implement a, 201
Stack class, 454, 466, 508, 513–515
methods, table of, 513
Stack frame, 446
Stack trace, 209–210, 216, 446
StackTraceElement class, 222, 446
methods, table of, 446
StandardOpenOption class, 635, 650, 651
enumeration, table of values for the, 637
StandardOpenOption.CREATE, 637, 645, 651
StandardOpenOption.READ, 637, 648, 650

StandardOpenOption.TRUNCATE_EXISTING, 637, 651

StandardOpenOption.WRITE, 637, 645, 651

Standard Template Library (STL), 455

start( ), 230, 233, 234, 235, 416, 422, 439, 690, 691, 692, 693, 695, 698, 741, 912, 915, 957, 959

startsWith( ), 380, 635

State enumeration, 256

Statements, 26

null, 90

Statements, control, 28

iteration, 81, 89–102

jump, 81, 102–108

selection, 81–89

static, 25, 145–146, 149, 313, 319, 320–321

member restrictions, 366

Static import, 14, 319–321
Stream(s)

- benefits, 628
- buffered, 599–603
- classes, byte, 290–291, 590–610
- classes, character, 290, 291–292, 590, 610–620
- closing, 589–590
- definition of, 290, 581
- filtered, 599, 628
- and NIO, 649–651
- predefined, 292–293

strictfp, 312

StrictMath class, 436

String class, 25, 58, 152–154, 371, 448, 563

- constructors, 372–374
String(s)

arrays of, 154

changing case of characters in, 387, 412, 413

comparison, 153, 378–382

concatenating, 152–153, 374–376, 385, 391–392

constants, 152

converting data into a, 376, 386–387

creating, 152, 372–374

extracting characters from, 377–378

formatted, creating a, 549–551

formatting a, 551, 556

immutable, 371, 389

length, obtaining, 153, 374

literals, 43–44, 374

modifying, 384–386

numbers to and from, converting, 410–411

as objects, 44, 58, 152, 371
parsing a formatted input, 525

reading, 294–296

searching, 382–383

StringBuffer class, 152, 371, 373, 384, 389–395, 448

StringBufferInputStream deprecated class, 582

StringBuilder class, 371, 373, 384, 395, 448, 549

and synchronization, 395

StringIndexOutOfBoundsException exception, 220

StringReader class, 292

StringTokenizer class, 525–526

methods, table of, 526

stringWidth( ), 766

StringWriter class, 292

Stroustrup, Bjarne, 6

Stubs (RMI), 925–926

Subclass, 20, 161, 162, 163, 179

subList( ), 459, 460
subMap( ), 484, 486

submit( ), 885

subSequence( ), 389, 395, 448

subSet( ), 461, 462, 473

substring( ), 384, 394

Sun Microsystems, 6, 14

super, 145

  and bounded wildcard arguments, 344
  and methods or instance variables, 170–171, 176

super( ), 323

  and superclass constructors, 167–170, 174

Superclass, 20, 161, 162, 163, 179, 187

  abstract, 181–184

Supplemental character, definition of, 414

@SuppressWarnings built-in annotation, 286, 288

suspend( ), 13, 251–254, 437, 443

Swing, 13, 289, 307, 309, 687, 735, 945–964, 965–992
applet, example of a simple, 957–959
application, example of a simple, 950–954
and the AWT, 735, 945, 946
component classes, list of, 948–949
components. See Components, Swing
download manager using, 1053–1077
event handling, 953–957
history of, 945–946
and MVC architecture, 947
packages, list of, 950
and painting, 957, 959–964
threading issues, 953–954, 957
Swing: A Beginner’s Guide (Schildt), 945
SwingConstants interface, 966
SwingUtilities class, 954
switch statement, 84–89
and auto-unboxing, 273
using enumeration constants to control a, 85, 260–261

using a String to control a, 15, 84–85, 87–88

versus the if statement, 88–89

Synchronization, 12, 229–230, 241–245

and atomic operations, 892–893

and collections, 465, 499, 508

and deadlock, 249–251, 253

and interprocess communication, 245–251

objects, using, 863–882

race condition and, 243

and StringBuilder class, 395

via synchronized block, 243–245, 499

via synchronized method, 230, 241–243

versus concurrency utilities, traditional, 861, 908

synchronized modifier, 241, 908

used with method, 241–243
used with object, 243–245

synchronizedList( ), 498, 499

synchronizedSet( ), 498, 499

Synchronizers, 862

SynchronousQueue class, 889

System class, 26, 34, 292, 423–426

   methods, table of, 423–424

System.console( ), 423, 620

System.err standard error stream, 292, 293

System.getProperties( ), 424, 519

System.getProperty( ), 424, 426

System.in standard input stream, 292, 293, 564, 620

System.in.read( ), 93

System.nanoTime( ), 900

System.out standard output stream, 26, 292, 293, 296, 297, 562, 605, 606, 620

   and static import, 321
Tabbed panes, 977–979

TableColumnModel, 990

TableModel, 990

TableModelEvent class, 990

tailMap( ), 484, 486

tailSet( ), 461, 462

tan( ), 433

tanh( ), 433

@Target built-in annotation, 286–287

TCP/IP, 12, 668

  client sockets, 671–675

  disadvantages of, 682

  server sockets, 671, 681–682

  See also Transmission Control Protocol (TCP)

TERMINATED, 256

Ternary if-then-else operator (?:), 75, 77–78
Text area, 794–796

Text components as an event source, 720

Text fields, 792–794, 1025

Swing, 967–968

Text formatting using java.text classes, 909, 927–930

Text output using font metrics, managing, 764–772

TextArea class, 737, 794–795

textChanged( ), 722

TextComponent class, 737, 792, 795

TextEvent class, 710, 718, 795

TextField class, 737, 792

TextListener interface, 720, 722

this, 124–125, 145

this( ), 321–323

Thompson, Ken, 4

Thread class, 13, 230–231, 232, 437–439, 544

constructors, 233, 236, 437
extending, 235–236

methods, table of, 438–439

**Thread(s)**

creating, 232–237
daemon, 897, 905

and deadlock, 249–251, 253, 437
definition of, 227

executors to manage, using, 863, 701–887
group, 232, 439–443

main, 231–232, 234–235, 236, 237, 238

messaging, 230, 245–250

pool, 883–885, 894

priorities, 229, 240–241, 437

resuming, 251–256, 441–443

states of, possible, 229, 256–257

stopping, 253–256

suspending, 230, 232, 251–256, 441–443
and Swing, event dispatching, 953–954, 957, 959
synchronization. See Synchronization

Thread.UncaughtExceptionHandler interface, 450

ThreadGroup class, 439–443, 450

methods, table of, 440–441

ThreadLocal class, 444

ThreadPoolExecutor class, 863, 883

throw, 207, 216–217

Throwable class, 208–209, 212, 216, 217, 221, 224, 306, 367, 445

methods defined by, table of, 222

obtaining object of the, 216–217

throws, 207, 217–218, 220

Thumb, 789

time, 531

Time and date

formatting, 552–554, 927–930
java.util classes that deal with, 530–539

timedJoin( ), 889

timedWait( ), 889

TIMED_WAITING, 256

Timer class, 544–546
    methods, table of, 545–546

TimerTask class, 544–546
    methods, table of, 545

Timestamp, event, 711

TimeUnit enumeration, 863, 869, 886, 888–889

TimeZone class, 536–537
    methods defined by, table of some, 536–537

to( ), 421

toAbsolutePath( ), 635, 652

toArray( ), 458, 459, 468–469

toBinaryString( ), 407, 409, 411

toCharArray( ), 378
toDays( ), 888

toDegrees( ), 436

toFile( ), 635

toHexString( ), 400, 402, 407, 409, 411

toHours( ), 888

Tokens, 525, 564

toLanguageTag( ), 539

toLowerCase( ), 387, 413, 414

Tomcat, 994, 995–996

toMicros( ), 888

toMillis( ), 888

toMinutes( ), 888

toNanos( ), 888

toOctalString( ), 407, 409, 411

TooManyListenersException, 936

toPath( ), 585, 635, 652

toRadians( ), 436
toSeconds( ), 888
totalMemory( ), 418–419
toUpperCase( ), 387, 413

transient modifier, 309, 937

translatePoint( ), 717

Transmission Control Protocol (TCP)

definition of, 667

and stream-based I/O, 668

See also TCP/IP

TreeExpansionEvent class, 987

TreeExpansionListener interface, 987

TreeMap class, 487, 492, 524

TreeModel, 987
  advantages to using, 590
tryLock( ), 863, 890
tryUnfork( ), 907
Two’s complement, 66–67
TYPE, 399, 403, 411, 414, 416
Type argument(s), 328, 330, 334
  and bounded types, 335–337
  and generic class hierarchies, 352
  and type inference, 359–361
Type conversion
  automatic, 35, 48, 130–131
  narrowing, 48
  widening, 48
Type enumeration, 421
Type interface, 451

Type parameter(s)

and bounded types, 334–336, 348–349

cannot create an instance of a, 365–366

and class hierarchies, 353–355

and erasure, 361–362, 366

and primitive types, 330

and static members, 366

and type safety, 330

used with a class, 327, 332, 334

used with a method, 328, 345, 346

Type safety

and arrays, 367

and collections, 455, 499, 505

and generic methods, 346

and generics, 325, 326, 329, 330–332, 455, 505, 506

and raw types, 349–352
and wildcard arguments, 337–339, 341

Typesafe view of a collection, obtaining a dynamically, 499

Type(s), 27

bounded, 335, 336

casting, 48–49, 50

checking, 10, 11, 35, 329, 330–332, 351, 367

class as a data, 109, 111, 113, 114, 126

inference, 359–361

parameterized, 326, 328

promotion, 37, 49–51, 69–70

raw. See Raw types

simple, 35

Types, primitive, 35–36, 114, 136, 268–269, 270, 275, 330

and collections, 456, 469

to a string representation, converting, 375, 376, 386
to or from a sequence of bytes, converting, 607, 608
wrappers for, 268–270, 275, 330, 398–416
TypeNotPresentException, 220

UDP protocol, 667, 668, 682
UI delegate, 947, 948
ulp( ), 434, 435
UnavailableException class, 999, 1002
Unboxing, 270
uncaughtException( ), 450
UncaughtExceptionHandler interface, 450
Unchecked warnings and raw types, 351
UnicastRemoteObject, 924
Unicode, 39, 40, 43, 290, 291, 373, 378, 388, 394, 395, 414, 610

code points, table of some Character methods providing support for, 415
support for 32–bit, 414

Unicode Technical Standard (UTS) 35, 539

Uniform Resource Identifier (URI), 681

Uniform Resource Locator (URL). See URL (Uniform Resource Locator)

UNIX, 4, 667

UnknownHostException, 669, 670

unlock( ), 863, 890

unmodifiable… collections methods, 499

Unreachable code, 213

unread( ), 602, 618

UnsupportedOperationException, 220, 457, 459, 477, 483, 499, 639

update( ), 541, 542, 693, 694, 695, 696, 749

  overriding, 693

URI (Uniform Resource Identifier), 681

URI class, 681
URL (Uniform Resource Locator), 675, 681, 993

specification format, 675

URL class, 675–676, 678, 679, 704

URLConnection class, 676–679

useDelimiter( ), 570–572

User Datagram Protocol (UDP), 667, 668, 682

useRadix( ), 573

UTS 35, 539

UUID class, 577

V

value (annotation member name), 285, 286

VALUE (PARAM NAME), 701

valueBound( ), 1006

valueChanged( ), 982, 983, 987, 988

valueOf( ), 262–263, 376, 386–387, 400, 402, 404, 405, 407, 409, 416, 447, 528

values( ), 262–263, 483, 484
valueUnbound( ), 1006

van Hoff, Arthur, 6

Varargs, 14, 155–160

  and ambiguity, 159–160

  methods, overloading, 158–159

  and Java’s printf( ), 155

  parameter, 157, 475

Variable(s), 44–47

  declaration, 27, 29, 44–45, 46

  definition of, 26, 44

  dynamic initialization of, 45

  enumeration, 260

final, 147

instance. See Instance variables

interface, 197, 204–205

object reference. See Object reference variables

scope and lifetime of, 45–47
Vector class, 454, 466, 482, 508, 509–513

legacy methods, table of, 510–511

VetoableChangeListener interface, 937, 938

Viewport, scroll pane, 980

visitFile( ), 658, 659

void, 25, 116

Void class, 416

volatile modifier, 310

VSPACE, 700, 701

W

wait( ), 185, 186, 245, 247–249, 254–255, 427, 890, 908

waitFor( ), 417, 420

WAITING, 256

WALL_TIME, 538

walkFileTree( ), 657–659

Warth, Chris, 6
Watchable interface, 634

WeakHashMap class, 487

Web browser
   executing applet in, 308, 309, 687, 691, 699, 739
   using status window of, 699

Web server and servlets, 993, 994

weightx constraint field, 807, 808

while loop, 89–90

Whitespace, 32, 82
   from a string, removing, 385–386

whois, 668, 672, 674, 675

WIDTH, 700

Wildcard arguments, 337–344
   bounded, 339–344
   used in creating an array, 367

Window
   displaying information within a, 749
as an event source, 720
frame. See Frame window fundamentals, 738–739
and graphics, 749
status, using, 699
Window class, 719, 737, 739, 816
windowActivated( ), 723
WindowAdapter class, 730
windowClosed( ), 723
windowClosing( ), 723, 740, 741
WindowConstants interface, 952
windowDeactivated( ), 723
windowDeiconified( ), 723
WindowEvent class, 710, 712, 718–719
WindowFocusListener interface, 720, 723
windowGainedFocus( ), 723
windowIconified( ), 723
WindowListener interface, 720, 723, 730, 740, 741

windowLostFocus( ), 723

windowOpened( ), 723

Work stealing, 897, 907

World Wide Web (WWW), 6, 7, 11, 675

wrap( ), 631

Wrappers, primitive type, 268–270, 275, 330, 398–416

WRITE, 421

write( ), 292, 296, 302, 592, 611, 623, 624, 633, 646, 647, 651, 663, 664

writeBoolean( ), 608, 624

writeDouble( ), 608, 624

Writer class, 291, 292, 590, 610, 613, 628

methods defined by, table of, 611–612

writeExternal( ), 623

writeInt( ), 608, 624
writeObject( ), 623
writeTo( ), 599

X

XOR (exclusive OR) operator (^)

bitwise, 66, 67, 68–69

Boolean logical, 75–76

Y

Yellin, Frank, 6

Z

Zero crossing, 67

ZIP file format, 579