Rogers Cadenhead

SEVENTH EDITION

Covers Java 8

Sams Teach Yourself Java in 21 Days

SAMS

About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Sams Teach Yourself Java™ in 21 Days

Rogers Cadenhead

800 East 96th Street, Indianapolis, Indiana 46240

Sams Teach Yourself Java™ in 21 Days

Copyright © 2016 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33710-9
ISBN-10: 0-672-33710-X

Library of Congress Control Number: 2015916217

Printed in the United States of America

First Printing December 2015

Editor-in-Chief
Mark Taub

Executive Editor
Mark Taber

Managing Editor
Kristy Hart

Project Editor
Elaine Wiley

Copy Editor
Barbara Hacha

Senior Indexer
Cheryl Lenser

Proofreader
Chuck Hutchinson

Technical Editor
Boris Minkin

Editorial Assistant
Vanessa Evans

Cover Designer
Mark Shirar

Senior Compositor
Gloria Schurick

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Contents at a Glance

Introduction

WEEK I: The Java Language

DAY 1 Getting Started with Java

DAY 2 The ABCs of Programming

DAY 3 Working with Objects

DAY 4 Lists, Logic, and Loops

DAY 5 Creating Classes and Methods

DAY 6 Packages, Interfaces, and Other Class Features

DAY 7 Exceptions and Threads

WEEK II: The Java Class Library

DAY 8 Data Structures

DAY 9 Working with Swing

DAY 10 Building a Swing Interface

DAY 11 Arranging Components on a User Interface

DAY 12 Responding to User Input

DAY 13 Creating Java2D Graphics

DAY 14 Developing Swing Applications

WEEK III: Java Programming

DAY 15 Working with Input and Output

DAY 16 Using Inner Classes and Closures

DAY 17 Communicating Across the Internet

DAY 18 Accessing Databases with JDBC 4.2 and Derby

DAY 19 Reading and Writing RSS Feeds

DAY 20 XML Web Services

DAY 21 Writing Android Apps with Java

APPENDIXES

A Using the NetBeans Integrated Development Environment

B This Book's Website

C Fixing a Problem with the Android Studio Emulator

D Using the Java Development Kit

E Programming with the Java Development Kit

Index

Table of Contents

Introduction

- How This Book Is Organized
- Who Should Read This Book
- Conventions Used in This Book

WEEK I: The Java Language

DAY 1: Getting Started with Java

- The Java Language
  - History of the Language
  - Introduction to Java
  - Selecting a Development Tool
- Object-Oriented Programming
- Objects and Classes
- Attributes and Behavior
  - Attributes of a Class of Objects
  - Behavior of a Class of Objects
  - Creating a Class
  - Running the Program
- Organizing Classes and Class Behavior
  - Inheritance
  - Creating a Class Hierarchy
  - Inheritance in Action
  - Interfaces
  - Packages
- Summary
- Q&A
- Quiz

- Questions
- Answers
- Certification Practice
- Exercises

DAY 2: The ABCs of Programming

- Statements and Expressions
- Variables and Data Types
  - Creating Variables
  - Naming Variables
  - Variable Types
  - Assigning Values to Variables
  - Constants
- Comments
- Literals
  - Number Literals
  - Boolean Literals
  - Character Literals
  - String Literals
- Expressions and Operators
  - Arithmetic
  - More About Assignment
  - Incrementing and Decrementing
  - Comparisons
  - Logical Operators
  - Operator Precedence
- String Arithmetic
- Summary
- Q&A
- Quiz

- Questions
- Answers
- Certification Practice
- Exercises

DAY 3: Working with Objects

- Creating New Objects
- Using `new`
- How Objects Are Constructed
- A Note on Memory Management
- Using Class and Instance Variables
- Getting Values
- Setting Values
- Class Variables
- Calling Methods
- Formatting Strings
- Nesting Method Calls
- Class Methods
- References to Objects
- Casting Objects and Primitive Types
- Casting Primitive Types
- Casting Objects
- Converting Primitive Types to Objects and Vice Versa
- Comparing Object Values and Classes
- Comparing Objects
- Determining the Class of an Object
- Summary
- Q&A
- Quiz
- Questions

Answers
Certification Practice
Exercises

DAY 4: **Lists, Logic, and Loops**
Arrays
Declaring Array Variables
Creating Array Objects
Accessing Array Elements
Changing Array Elements
Multidimensional Arrays
Block Statements
If Conditionals
Switch Conditionals
The Ternary Operator
For Loops
While and Do Loops
While Loops
Do-While Loops
Breaking Out of Loops
Labeled Loops
Summary
Q&A
Quiz
Questions
Answers
Certification Practice
Exercises

DAY 5: **Creating Classes and Methods**
Defining Classes

Creating Instance and Class Variables
- Defining Instance Variables
- Class Variables

Creating Methods
- Defining Methods
- The `this` Keyword
- Variable Scope and Method Definitions
- Passing Arguments to Methods
- Class Methods

Creating Java Applications
- Helper Classes

Java Applications and Arguments
- Passing Arguments to Java Applications
- Handling Arguments in Your Java Application

Creating Methods with the Same Name

Constructors
- Basic Constructors
- Calling Another Constructor
- Overloading Constructors

Overriding Methods
- Creating Methods That Override Existing Methods
- Calling the Original Method
- Overriding Constructors

Summary

Q&A

Quiz
- Questions
- Answers

Certification Practice

Exercises

DAY 6: **Packages, Interfaces, and Other Class Features**

- Modifiers
  - Access Control for Methods and Variables
- Static Variables and Methods
- Final Classes, Methods, and Variables
  - Variables
  - Methods
  - Classes
- Abstract Classes and Methods
- Packages
  - The `import` Declaration
  - Class Name Conflicts
- Creating Your Own Packages
  - Picking a Package Name
  - Creating the Folder Structure
  - Adding a Class to a Package
  - Packages and Class Access Control
- Interfaces
  - The Problem of Single Inheritance
  - Interfaces and Classes
  - Implementing and Using Interfaces
  - Implementing Multiple Interfaces
  - Other Uses of Interfaces
- Creating and Extending Interfaces
  - New Interfaces
  - Methods Inside Interfaces
  - Extending Interfaces
  - Creating an Online Storefront
- Summary
- Q&A

- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 7: Exceptions and Threads

- Exceptions
  - Exception Classes
- Managing Exceptions
  - Exception Consistency Checking
  - Protecting Code and Catching Exceptions
  - The `finally` Clause
- Declaring Methods That Might Throw Exceptions
  - The `throws` Clause
  - Which Exceptions Should You Throw?
  - Passing on Exceptions
  - `throws` and Inheritance
- Creating and Throwing Exceptions
  - Throwing Exceptions
  - Creating Your Own Exceptions
  - Combining `throws`, `try`, and `throw`
- When Not to Use Exceptions
  - Bad Style Using Exceptions
- Threads
  - Writing a Threaded Program
  - A Threaded Application
  - Stopping a Thread
- Summary
- Q&A

- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

WEEK II: The Java Class Library

DAY 8: Data Structures

- Moving Beyond Arrays
- Java Structures
  - `Iterator`
  - Bit Sets
  - Array Lists
  - Looping Through Data Structures
  - Stacks
  - `Map`
  - Hash Maps
- Generics
- Enumerations
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 9: Working with Swing

- Creating an Application
  - Creating an Interface
  - Developing a Framework

- Creating a Component
- Adding Components to a Container
- Working with Components
- Image Icons
- Labels
- Text Fields
- Text Areas
- Scrolling Panes
- Check Boxes and Radio Buttons
- Combo Boxes
- Lists
- The Java Class Library
- Summary
- Q&A
- Quiz
- Questions
- Answers
- Certification Practice
- Exercises

DAY 10: Building a Swing Interface

- Swing Features
- Standard Dialog Boxes
- Using Dialog Boxes
- Sliders
- Scroll Panes
- Toolbars
- Progress Bars
- Menus
- Tabbed Panes

- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 11: Arranging Components on a User Interface

- Basic Interface Layout
  - Laying Out an Interface
  - Flow Layout
  - Box Layout
  - Grid Layout
  - Border Layout
- Mixing Layout Managers
- Card Layout
  - Using Card Layout in an Application
  - Cell Padding and Insets
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 12: Responding to User Input

- Event Listeners
  - Setting Up Components
  - Event-Handling Methods

- Working with Methods
  - Action Events
  - Focus Events
  - Item Events
  - Key Events
  - Mouse Events
  - Mouse Motion Events
  - Window Events
  - Using Adapter Classes
  - Using Inner Classes
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 13: Creating Java2D Graphics

- The Graphics2D Class
  - The Graphics Coordinate System
- Drawing Text
  - Improving Fonts and Graphics with Antialiasing
  - Finding Information About a Font
- Color
  - Using `Color` Objects
  - Testing and Setting the Current Colors
- Drawing Lines and Polygons
  - User and Device Coordinate Spaces
  - Specifying the Rendering Attributes

- Creating Objects to Draw
- Drawing Objects
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 14: Developing Swing Applications

- Java Web Start
- Using Java Web Start
  - Creating a JNLP File
  - Supporting Web Start on a Server
  - Additional JNLP Elements
- Improving Performance with SwingWorker
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

WEEK III: Java Programming

DAY 15: Working with Input and Output

- Introduction to Streams
  - Using a Stream
  - Filtering a Stream
  - Handling Exceptions

Byte Streams
File Streams
Filtering a Stream
Byte Filters
Character Streams
Reading Text Files
Writing Text Files
Files and Paths
Summary
Q&A
Quiz
Questions
Answers
Certification Practice
Exercises

DAY 16: **Using Inner Classes and Closures**
Inner Classes
Anonymous Inner Classes
Closures
Summary
Q&A
Quiz
Questions
Answers
Certification Practice
Exercises

DAY 17: **Communicating Across the Internet**
Networking in Java
Opening a Stream Over the Net

- Sockets
- Socket Servers
- Testing the Server
- The `java.nio` Package
- Buffers
- Channels
- Summary
- Q&A
- Quiz
- Questions
- Answers
- Certification Practice
- Exercises

DAY 18: Accessing Databases with JDBC 4.2 and Derby

- Java Database Connectivity
- Database Drivers
- Examining a Database
- Reading Records from a Database
- Writing Records to a Database
- Moving Through Resultsets
- Summary
- Q&A
- Quiz
- Questions
- Answers
- Certification Practice
- Exercises

DAY 19: Reading and Writing RSS Feeds

- Using XML

- Designing an XML Dialect
- Processing XML with Java
- Processing XML with XOM
  - Creating an XML Document
  - Modifying an XML Document
  - Formatting an XML Document
  - Evaluating XOM
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 20: XML Web Services

- Introduction to XML-RPC
- Communicating with XML-RPC
  - Sending a Request
  - Responding to a Request
- Choosing an XML-RPC Implementation
- Using an XML-RPC Web Service
- Creating an XML-RPC Web Service
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

DAY 21: Writing Android Apps with Java

- The History of Android
- Writing an Android App
  - Organizing an Android Project
  - Creating the Program
- Running the App
- Designing an Android App
  - Preparing Resources
  - Configuring a Manifest File
  - Designing the Graphical User Interface
  - Writing Code
- Summary
- Q&A
- Quiz
  - Questions
  - Answers
- Certification Practice
- Exercises

APPENDIXES

APPENDIX A: Using the NetBeans Integrated Development Environment

- Installing NetBeans
- Creating a New Project
- Creating a New Java Class
- Running the Application
- Fixing Errors
- Expanding and Shrinking a Pane
- Exploring NetBeans

APPENDIX B: This Book's Website

APPENDIX C: Fixing a Problem with the Android Studio Emulator

- Problems Running an App
  - Install HAXM in Android Studio
  - Install HAXM on Your Computer
  - Checking BIOS Settings

APPENDIX D: Using the Java Development Kit

- Choosing a Java Development Tool
  - Installing the Java Development Kit
- Configuring the Java Development Kit
  - Using a Command-Line Interface
  - Opening Folders in MS-DOS
  - Creating Folders in MS-DOS
  - Running Programs in MS-DOS
  - Correcting Configuration Errors
- Using a Text Editor
- Creating a Sample Program
  - Compiling and Running the Program in Windows
- Setting Up the `CLASSPATH` Variable
  - Setting the `Classpath` on Most Windows Versions
  - Setting the `CLASSPATH` on Windows 98 or Me

APPENDIX E: Programming with the Java Development Kit

- Overview of the JDK
- The `java` Virtual Machine
- The `javac` Compiler
- The `appletviewer` Browser
- The `javadoc` Documentation Tool
- The `jar` Java File Archival Tool
- The `jdb` Debugger
  - Debugging Applications

Debugging Applets
Advanced Debugging Commands
Using System Properties
The `keytool` and `jarsigner` Code Signing Tools
Index

About the Author

Rogers Cadenhead is a programmer and author. He has written more than 30 books on programming and web publishing, including *Sams Teach Yourself Java in 24 Hours* and *Absolute Beginner's Guide to Minecraft Mods Programming*. He also publishes the Drudge Retort and other websites that receive more than 20 million visits a year. He maintains this book's official website at www.java21days.com and a personal weblog at http://workbench.cadenhead.org.

Dedication

To my son Max Cadenhead, who just began his freshman year at art school.
Your talent and dedication to your craft at such an early age makes me proud.

—Dad

Acknowledgments

A book of this scope (and heft!) requires the hard work and dedication of numerous people. Most of them are at Sams Publishing in Indianapolis, and to them I owe considerable thanks—in particular, to Boris Minkin, Barbara Hacha, Elaine Wiley, and Mark Taber. Most of all, thanks to my wife, Mary, and my sons, Max, Eli, and Sam.

I'd also like to thank readers who have sent helpful comments about corrections, typos, and suggested improvements regarding this book and its prior editions. The list includes the following people: Dave Barton, Patrick Benson, Ian Burton, Lawrence Chang, Jim DeVries, Ryan Esposto, Kim Farr, Sam Fitzpatrick, Bruce Franz, Owen Gailar, Rich Getz, Bob Griesemer, Jenny Guriel, Brenda Henry-Sewell, Ben Hensley, Jon Hereng, Drew Huber, John R. Jackson, Bleu Jaegel, Natalie Kehr, Mark Lehner, Stephen Loscialpo, Brad Kaenel, Chris McGuire, Paul Niedenzu, E.J. O'Brien, Chip Pursell, Pranay Rajgarhia, Peter Riedlberger, Darrell Roberts, Luke Shulenburger, Mike Tomsic, John Walker, Joseph Walsh, Mark Weiss, P.C. Whidden, Chen Yan, Kyu Hwang Yeon, and J-F. Zurcher.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: errata@informit.com

Mail: Addison-Wesley/Prentice Hall Publishing
ATTN: Reader Feedback
330 Hudson Street
7th Floor
New York, New York, 10013

Reader Services

Visit our website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

Some revolutions catch the world by surprise. Twitter, the Linux operating system, and *Pawn Stars* all rose to prominence unexpectedly.

The remarkable success of the Java programming language, on the other hand, caught nobody by surprise. Java has been a source of great expectations since its introduction 20 years ago. When Java was introduced in web browsers, a torrent of publicity welcomed the arrival of the new language.

Sun Microsystems cofounder Bill Joy proclaimed, "This represents the end result of nearly 15 years of trying to come up with a better programming language and environment for building simpler and more reliable software."

Sun, which created Java in 1991 and first released it to the public four years later, was acquired by Oracle in 2010. Oracle, which has been committed to Java development since its earliest years, has continued to support the language and produce new versions.

In the ensuing years, Java lived up to a considerable amount of its hype. The language has become as strong a part of software development as the beverage of the same name. One kind of Java keeps programmers up nights. The other kind enables programmers to rest easier after they have developed their software.

Java was originally offered as a technology for enhancing websites with programs that run in browsers. Today, it's more likely to be found on servers, driving dynamic web applications backed by relational databases on some of the web's largest sites, and on millions of Android cell phones and tablets running popular apps such as Clash of Clans and Instagram.

Each new release of Java strengthens its capabilities as a general-purpose programming language for a wide range of environments. Today, Java is being put to use in desktop applications, Internet servers, mobile devices, and many other environments. It's even making a comeback in the browser with sophisticated applications created in Java.

Now in its ninth major release—Java 8—the Java language has matured into a full-featured competitor to other general-purpose development languages, such as C++, Python, and Ruby.

You might be familiar with Java programming tools such as Eclipse, NetBeans, and IntelliJ IDEA. These programs make it possible to develop functional Java programs, and you also can use Oracle's Java Development Kit. The kit, which is available for free on the Web, is a set of command-line tools for writing, compiling, and testing Java programs. NetBeans, another free tool offered by Oracle, is an integrated development environment for the creation of Java programs. It can be downloaded from www.netbeans.org.

This book introduces you to all aspects of Java software development using the most current version of the language and the best available techniques in Java Standard Edition, the most widely used version of the language. Programs are prepared and tested using NetBeans, so you can quickly demonstrate the skills you master each day.

Reading this book will help you understand why Java has become the most widely employed programming language on the planet.

How This Book Is Organized

Sams Teach Yourself Java in 21 Days teaches you about the Java language and how to use it to create applications for any computing environment and Android apps that run on cell phones and other mobile devices. By the time you have finished the book, you'll have well-rounded knowledge of Java and the Java class libraries. Using your new skills, you will be able to develop your own programs for tasks such as web services, database connectivity, XML processing, and mobile programming.

You learn by doing in this book, creating several programs each day that demonstrate the topics being introduced. The source code for all these programs is available on the book's official website at www.java21days.com, along with other supplemental material such as answers to reader questions.

This book covers the Java language and its class libraries in 21 days, organized into three weeks. Each week covers a broad area of developing Java programs.

In the first week, you learn about the Java language itself:

- Day 1 covers the basics—what Java is, why you should learn the language, and how to create software using a powerful style of

development called object-oriented programming. You create your first Java application.

- On Day 2, you dive into the fundamental Java building blocks—data types, variables, and expressions.
- Day 3 goes into detail about how to deal with objects in Java—how to create them, use their variables, call their methods, and compare them.
- On Day 4, you give Java programs some brainpower using conditionals and work with arrays and loops.
- Day 5 fully explores creating classes—the basic building blocks of any Java program.
- On Day 6, you discover more about interfaces and packages, which are useful for grouping classes and organizing a class hierarchy.
- Day 7 covers three powerful features of Java: exceptions, the ability to deal with errors; threads, the capability to run different parts of a program simultaneously; and assertions, a technique for making programs more reliable.

Week 2 is dedicated to the most useful classes offered by Oracle for use in your own Java programs:

- Day 8 introduces data structures that you can use as an alternative to strings and arrays—array lists, stacks, maps, hash maps, and bit sets. It also describes a special `for` loop that makes them easier to use.
- Day 9 begins a five-day exploration of visual programming. You learn how to create a graphical user interface using Swing classes for interfaces, graphics, and user input.
- Day 10 covers more than a dozen interface components you can use in a Java program, including buttons, text fields, sliders, scrolling text areas, and icons.
- Day 11 explains how to make a user interface look marvelous using *layout managers*, a set of classes that determine how components on an interface are arranged.
- Day 12 concludes the coverage of Swing with event-handling classes, which enable a program to respond to mouse clicks and other user interactions.

- On Day 13, you learn about drawing shapes and characters on user interface components.
- Day 14 demonstrates how to use Java Web Start, a technique that makes installing a Java program as easy as clicking a web page link.

Week 3 moves into advanced topics:

- Day 15 covers input and output using *streams*, a set of classes that enable file access, network access, and other sophisticated data handling.
- Day 16 provides a complete introduction to closures, the most exciting new feature of Java 8. Also called *lambda expressions*, closures make it possible to employ a new type of coding called functional programming in Java for the first time. Inner classes are explored in greater depth as they relate to closures.
- On Day 17, you extend your knowledge of streams to write programs that communicate with the Internet, including socket programming, buffers, channels, and URL handling.
- Day 18 shows you how to connect to relational databases using Java Database Connectivity (JDBC) version 4.2. You learn how to exploit the capabilities of Derby, the open source database that's included with Java.
- Day 19 covers how to read and write RSS documents using the XML Object Model (XOM), an open source Java class library. RSS feeds, one of the most popular XML dialects in use today, enable millions of people to follow site updates and other new web content.
- Day 20 explores how to write web services clients with the language and the Apache XML-RPC class library.
- Day 21 covers the fastest-growing area of Java programming: developing apps for Android phones and mobile devices. Using Google's free Android Studio as a development environment and a free Android development kit, you create apps that can be deployed and tested on a phone.

Who Should Read This Book

This book teaches the Java language to three groups:

- Novices who are relatively new to programming
- People who have been introduced to earlier versions of Java
- Experienced developers in other languages, such as Visual C++, Visual Basic, or Python

When you're finished with this book, you'll be able to tackle any aspect of the Java language. You'll also be comfortable enough to tackle your own ambitious programming projects, both on and off the Web.

If you're somewhat new to programming or have never written a program, you might wonder whether this is the right book for you. Because all the concepts in this book are illustrated with working programs, you'll be able to work your way through the subject regardless of your experience level. If you understand what variables and loops are, you'll be able to benefit from this book. You might want to read this book if any of the following are true:

- You had some beginning programming lessons in school, you grasp what programming is, and you've heard that Java is easy to learn, powerful, and cool.
- You've programmed in another language for a few years, you keep hearing accolades for Java, and you want to see whether it lives up to its hype.
- You've heard that Java is great for web application and Android programming.

If you've never been introduced to object-oriented programming, which is the style of programming that Java embodies, don't be discouraged. This book assumes that you have no background in object-oriented design. You'll get a chance to learn this development methodology as you're learning Java.

If you're a complete beginner to programming, this book might move a little fast for you. Java is a good language to start with, though, and if you take it slowly and work through all the examples, you can still pick up Java and start creating your own programs.

Conventions Used in This Book

Note

A Note presents an interesting, sometimes technical, piece of information related to the discussion.

Tip

A Tip offers advice, such as an easier way to do something.

Caution

A Caution advises you of potential problems and helps you steer clear of disaster.

Text that you type and text that appears onscreen is presented in a `monospace` font:

Click here to view code image

```
Monospace looks like this. Hi, mom!
```

This font represents how text looks onscreen. Placeholders for variables and expressions appear in *`monospace italic`*.

The end of each lesson offers several special features: answers to commonly asked questions about that day's subject matter, a quiz to test your knowledge of the material, two exercises that you can try on your own, and a practice question in case you're preparing for Java certification. Solutions to the exercises and the answer to the certification question can be found on the book's official website at www.java21days.com.

Week I: The Java Language

1 Getting Started with Java

2 The ABCs of Programming

3 Working with Objects

4 Lists, Logic, and Loops

5 Creating Classes and Methods

6 Packages, Interfaces, and Other Class Features

7 Exceptions and Threads

Day 1. Getting Started with Java

> *The thing that Java tries to do and is actually remarkably successful at is spanning a lot of different domains, so you can do app server work, you can do cell phone work, you can do scientific programming, you can write software, do interplanetary navigation, all kinds of stuff...*
>
> —Java language creator James Gosling

When the Java programming language was unleashed on the public in 1995, it was an inventive toy for the Web that had the potential to be more.

The word "potential" is a compliment that comes with an expiration date. Sooner or later, potential must be realized, or new words and phrases are used in its place, such as "slacker," "letdown," "waste," or "major disappointment to your mother and me."

As you develop your skills in this book's 21 one-day tutorials, you'll be in a good position to judge whether the language has lived up to more than a decade of hype.

You'll also become a Java programmer with a lot of potential.

The Java Language

Now in its ninth major release, Java has lived up to the expectations that accompanied its arrival. More than four million programmers have learned the language and are using it in places such as NASA, IBM, Kaiser Permanente, and Google. It's a standard part of the academic curriculum at many computer science departments around the world. First used to create simple programs on web pages, Java can be found today in the following places (and many more):

- Web servers
- Relational databases
- Orbiting telescopes
- E-book readers
- Cell phones

Although Java remains useful for web developers, its ambitions today extend far beyond the Web. Java has matured into one of the most popular general-purpose programming languages.

History of the Language

The story of the Java language is well known by this point. James Gosling and a team of developers were working on an interactive TV project at Sun Microsystems in the mid-1990s when Gosling became frustrated with the language being used. C++ was an object-oriented programming language developed a decade earlier as an extension of the C language.

To address some of the things that frustrated him about C++, Gosling holed up in his office and created a new language that was suitable for his project.

Although that interactive TV effort flopped, Gosling's language had unforeseen applicability to a new medium that was becoming popular at the same time: the Web.

Java was released to the public for the first time in 1995. Although most of the language's features were primitive compared with C++ (and Java today), special Java programs called applets could be run as part of web pages on the most popular web browser at that time, Netscape Navigator.

This functionality—the first interactive programming available on the Web—drew so much attention to the new language that several hundred thousand programmers learned Java in its first six months.

Even after the novelty of Java web programming wore off, the overall benefits of the language became clear, and the programmers stuck around. There are more professional Java programmers today than C++ programmers.

Sun Microsystems controlled the development of the Java language from its inception until 2010, when the company was acquired by the database and enterprise software giant Oracle in a $7.4 billion deal. Oracle, a longtime user of the language on its own products, has a strong commitment to supporting Java and continues to increase its capabilities with each new release.

Introduction to Java

Java is an object-oriented, platform-neutral, secure language designed to be easier to learn than C++ and harder to misuse than C and C++.

Object-oriented programming (OOP) is a software development methodology in which a program is conceptualized as a group of objects that work together. Objects are created from templates called *classes*, and they contain data and the statements required to use that data. Java is primarily object-oriented, as you'll see later today when you create your first class and use it to create objects.

Platform neutrality is a program's ability to run without modification in different computing environments. Java programs are transformed into a format called *bytecode* that can be run by any computer or device equipped with a Java Virtual Machine (JVM). You can create a Java program on a Windows 10 machine that runs on a Linux web server, an Apple Mac using OS 10.10, and a Samsung Android phone. As long as a platform has a JVM, it can run the bytecode.

Although the relative ease of learning one language over another is always a point of contention among programmers, Java was designed to be easier than C++ primarily in the following ways:

- Java automatically takes care of memory allocation and deallocation, freeing programmers from this error-prone and complex task.
- Java doesn't include pointers, a powerful feature for experienced programmers that can be easily misused and introduce major security vulnerabilities.
- Java includes only single inheritance in object-oriented programming.

The lack of pointers and the presence of automatic memory management are two key elements of Java security.

Selecting a Development Tool

Now that you've been introduced to Java as a spectator, it's time to put some of these concepts into play and create your first Java program.

If you work your way through the 21 days of this book, you'll become well versed in Java's capabilities, including graphics, file input and output, XML processing, and Android app development. You will write programs that run

on web pages and others that run on your computer, web servers, or other computing environments.

Before you get started, you must have software on your computer that can be used to edit, prepare, and run Java programs that use the most up-to-date version of the language: Java 8.

Several popular integrated development environments (IDEs) for Java support version 8, including IntelliJ IDEA and the open source software Eclipse.

If you are learning to use these tools at the same time as you are learning Java, it can be a daunting task. Most IDEs are aimed primarily at experienced programmers who want to be more productive, not new people who are taking their first foray into a new language.

The simplest tool for Java development is the Java Development Kit, which is free and can be downloaded from www.oracle.com/technetwork/java/javase/downloads.

Whenever Oracle releases a new version of Java, it also makes a free development kit available over the Web to support that version. The current release is Java SE Development Kit 8.

The drawback of developing Java programs with the JDK is that it is a set of command-line tools. Therefore, it has no graphical user interface for editing programs, turning them into Java classes, and testing them. (A command line is simply a prompt for typing text commands. It's available in Windows as the program Command Prompt.)

Oracle offers an excellent free IDE for Java programmers called NetBeans on the website www.netbeans.org. Because NetBeans is easier to use for most people than the JDK, it's employed throughout this book.

If you don't have a Java development tool on your computer yet and you want to try NetBeans, you can find out how to get started with the software in Appendix A, "Using the NetBeans Integrated Development Environment." The appendix covers how to download and install the kit and use it to create a sample Java program to make sure it works.

As soon as you have a Java development tool on your computer that supports Java 8, you're ready to dive into the language.

If you don't have one on your computer yet, now's the time to set one up—preferably NetBeans.

Tip

For more information on the other IDEs for Java, visit the IDEA site at www.jetbrains.com/idea and Eclipse at www.eclipse.org. The IDE Android Studio, a version of IntelliJ IDEA, is used for creating mobile apps in Day 21, "Writing Android Apps with Java."

Object-Oriented Programming

The biggest challenge for a new Java programmer is learning object-oriented programming while learning the Java language.

Although this might sound daunting if you are unfamiliar with this style of programming, think of it as a two-for-one discount for your brain. You will learn object-oriented programming by learning Java. There's no other way to make use of the language.

Object-oriented programming is an approach to building computer programs that mimics how objects are assembled in the physical world.

By using this style of development, you can create programs that are more reusable, reliable, and understandable.

To get to that point, you first must explore how Java embodies the principles of object-oriented programming.

If you already are familiar with object-oriented programming, much of today's material will be a review for you. Even if you skim over the introductory material, you should create the sample program to get some experience in developing, compiling, and running Java programs.

There are many ways to conceptualize a computer program. One way is to think of a program as a series of instructions carried out in sequence, which commonly is called *procedural programming*. Some programmers start by learning a procedural language such as a version of BASIC.

Procedural languages mirror how a computer carries out instructions, so the programs you write are tailored to the computer's manner of doing things. One of the first things a procedural programmer must learn is how to break a problem into a series of simple steps followed in order.

Object-oriented programming looks at a computer program from a different angle, focusing on the task the program was created to perform, not on how a

computer handles tasks.

In object-oriented programming, a computer program is conceptualized as a set of objects that work together to accomplish a task. Each object is a separate part of the program, interacting with the other parts in highly controlled ways.

For a real-life example of object-oriented design, consider a stereo system. Most systems are built by hooking together a bunch of different objects, which are more commonly called components. If you came back from a stereo shopping trip, you might bring home all these objects:

- Speaker components that play midrange and high-frequency sounds.
- A subwoofer component that plays low bass frequency sounds.
- A tuner component that receives radio broadcast signals.
- A CD player component that reads audio data from CDs.
- A turntable component that reads audio data from vinyl records.

These components are designed to interact with each other using standard input and output connectors. Even if you bought speakers, subwoofer, tuner, CD player, and turntable made by different companies, you could combine them to form a stereo system—as long as each component has standard connectors.

Object-oriented programming works under the same principle: You put together a program by creating new objects and connecting them to each other and to existing objects provided by Oracle or another software developer. Each object is a component in the larger program, and they are combined together in a standard way. Each object plays a specific role in the larger program.

An *object* is a self-contained element of a computer program that represents a related group of features and that is designed to accomplish specific tasks.

Objects and Classes

Object-oriented programming is modeled on the observation that in the physical world, objects are made up of many kinds of smaller objects.

The capability to combine objects is only one aspect of object-oriented programming. Another important feature is the use of classes.

A *class* is a template used to create an object. Every object created from the same class has similar features.

Classes embody all features of a particular set of objects. When you write a program in an object-oriented language, you don't define individual objects. Instead, you define classes used to create those objects.

If you were writing a networking program in Java, you could create a `HighSpeedModem` class that describes the features of all Internet modems. These devices have the following common features:

- They connect to a computer's ethernet port.
- They send and receive information.
- They communicate with Internet servers.

The `HighSpeedModem` class serves as an abstract model for the concept of such a modem. To have something concrete you can manipulate in a program, you need an object. You must use the `HighSpeedModem` class to create a `HighSpeedModem` object. The process of creating an object from a class is called *instantiation*, which is why objects also are called *instances*.

A `HighSpeedModem` class can be used to create different `HighSpeedModem` objects in a program, each with different features, such as the following:

- Some function as a wireless Internet gateway, whereas others do not.
- Some can be used as a network router.
- They support different connection speeds.

Even with these differences, two `HighSpeedModem` objects still have enough in common to be recognizable as related objects.

Here's another example: Using Java, you could create a class to represent all command buttons—the clickable rectangles that appear on windows, dialogs, and other parts of a program's graphical user interface.

When the `CommandButton` class is developed, it could define these features:

- The text displayed on the button
- The size of the button
- Aspects of its appearance, such as whether it has a 3D shadow

The `CommandButton` class also could define how a button behaves when it is clicked.

After you define the `CommandButton` class, you can create instances of that button—in other words, `CommandButton` objects. The objects all take on the basic features of a button as defined by the class. But each one could have a different appearance and slightly different behavior, depending on what you need that object to do.

By creating a `CommandButton` class, you don't have to keep rewriting the code for each button you want to use in your programs. In addition, you can reuse the `CommandButton` class to create different kinds of buttons as you need them, both in this program and in others.

When you write a Java program, you design and construct a set of classes. When your program runs, objects are created from those classes and used as needed. Your task as a Java programmer is to create the right set of classes to accomplish what your program needs to accomplish.

Fortunately, you don't have to start from scratch. The Java language includes the Java Class Library, more than 4,000 classes that implement most of the functionality you will need. These classes are installed along with a development tool such as the JDK.

When you're talking about programming in the Java language, you're actually talking about using this class library and some standard keywords and operators defined in Java.

The class library handles numerous tasks, such as mathematical functions, text, graphics, user interaction, and networking. Working with these classes is no different from working with the Java classes you create.

For complicated Java programs, you might create a whole set of new classes that form their own class library for use in other programs.

Reuse is one of the fundamental benefits of object-oriented programming.

Note

In the Java Class Library, one of Java's standard classes, `JButton` in the `javax.swing` package, encompasses all the functionality of this hypothetical `CommandButton` example, along with a lot more. You'll get a chance to create objects from this class during Day 9, "Working with Swing."

Attributes and Behavior

A Java class consists of two distinct types of information: attributes and behavior.

Both of these are present in `MarsRobot`, a project you will implement today as a class. This project, a simple simulation of a planetary exploration vehicle, is inspired by the Mars Exploration Rovers used by NASA's Jet Propulsion Laboratory program to do research on the surface and geology of the planet Mars.

Before you create the program, you need to learn some things about how object-oriented programs are designed in Java. The concepts may be difficult to understand as you're introduced to them, but you'll get plenty of practice with them throughout the book.

Attributes of a Class of Objects

Attributes are the data that differentiate one object from another. They can be used to determine the appearance, state, and other qualities of objects that belong to that class.

An exploration vehicle could have the following attributes:

- **Status**—Exploring, moving, returning home
- **Speed**—Measured in miles per hour
- **Temperature**—Measured in degrees Fahrenheit

In a class, attributes are defined by *variables*—places to store information in a computer program. *Instance variables* are attributes that have values that differ from one object to another.

An instance variable defines an attribute of one particular object. The object's class defines what kind of attribute it is, and each instance stores its

own value for that attribute. Instance variables also are called *object variables* or *member variables*.

Each class attribute has a single corresponding variable. You change that attribute of the object by changing the value of the variable.

For example, the `MarsRobot` class defines a `speed` instance variable. This must be an instance variable because each robot travels at a different speed. The value of a robot's `speed` instance variable could be changed to make the robot move more quickly or slowly.

Instance variables can be given a value when an object is created and then stay constant throughout the life of the object. They also can be given different values as the object is used in a running program.

For other variables, it makes more sense to have one value that is shared by all objects of that class. These attributes are called *class variables*.

A class variable defines an attribute of an entire class. The variable applies to the class itself and to all its instances, so only one value is stored, no matter how many objects of that class have been created.

An example of a class variable for the `MarsRobot` class would be a `topSpeed` variable that holds the maximum speed any robot is capable of traveling. If an instance variable were created to hold the speed, each object could have a different value for this variable. That could cause problems because no robot is capable of exceeding it.

Using a class variable prevents this problem because all objects of that class share the same value automatically. Each `MarsRobot` object would have access to that variable.

Behavior of a Class of Objects

Behavior refers to the things that a class of objects can do—both to themselves and to other objects. Behavior can be used to change an object's attributes, receive information from other objects, and send messages to other objects, asking them to perform tasks.

A Mars robot could have the following behavior:

- Check the current temperature
- Begin a survey
- Accelerate or decelerate its speed

- Report its current location

Behavior for a class of objects is implemented using methods.

Methods are groups of related statements in a class that perform a specific task. They are used to accomplish specific tasks on their own objects and on other objects and are comparable to functions and subroutines in other programming languages. A well-designed method performs only one task.

Objects communicate with each other using methods. A class or object can call methods in another class or object for many reasons, including the following:

- To report a change to another object
- To tell the other object to change something about itself
- To ask another object to do something

For example, two Mars robots could use methods to report their locations to each other and avoid collisions, and one robot could tell another to stop so that it can pass by safely.

Just as there are instance and class variables, there also are instance and class methods. Instance methods, which are usually just called methods, are used when you are working with an object of the class. If a method changes an individual object, it must be an instance method. Class methods apply to a class itself.

Creating a Class

To see classes, objects, attributes, and behavior in action, you will develop a `MarsRobot` class, create objects from that class, and work with them in a running program.

Note

The main purpose of this project is to explore object-oriented programming. You'll learn more about Java programming syntax during Day 2, "The ABCs of Programming."

This book uses NetBeans as its primary development tool for creating Java programs. NetBeans organizes Java classes into projects. It will be useful to

have a project to hold the classes you create in this book. If you have not done so already, create a project:

1. Choose the menu command File, New Project. The New Project dialog appears.
2. In the Categories pane, choose Java.
3. In the Projects pane, choose Java Application and click Next. The New Java Application dialog opens.
4. In the Project Name text field, enter the name of the project (I used `Java21`). The Project Folder field is updated as you type the name. Make a note of this folder—it's where your Java programs can be found on your computer.
5. Deselect the check box Create Main Class.
6. Click Finish.

The project is created. You can use it throughout the book for the programs you work on.

If you created a project earlier, it probably will be open in NetBeans. (If not, choose the menu command File, Open Recent Project to select it.) A new class you create will be added to this project.

To begin your first class, run NetBeans and start a new program:

1. Choose the menu command File, New File. The New File dialog opens.
2. In the Categories pane, choose Java.
3. In the File Types pane, choose Empty Java File and click Next. The Empty Java File dialog opens.
4. In the Class Name text field, enter `MarsRobot`. The file you're creating is shown in the Created File field, which can't be edited. This file has the name `MarsRobot.java`.
5. Click Finish.

The NetBeans source code editor opens with nothing in it. Fill it with the code in Listing 1.1. When you're done, save the file using the menu command File, Save. The file `MarsRobot.java` will be saved.

Note

Don't type the numbers at the beginning of each line in the listing. They're not part of the program. They are included so that individual lines can be described for instructive purposes in this book.

LISTING 1.1 The Full Text of `MarsRobot.java`.

Click here to view code image

```
 1: class MarsRobot {
 2:     String status;
 3:     int speed;
 4:     float temperature;
 5:
 6:     void checkTemperature() {
 7:         if (temperature < -80) {
 8:             status = "returning home";
 9:             speed = 5;
10:         }
11:     }
12:
13:     void showAttributes() {
14:         System.out.println("Status: " + status);
15:         System.out.println("Speed: " + speed);
16:         System.out.println("Temperature: " + temperature);
17:     }
18: }
```

When you save this file, if it has no errors, NetBeans automatically creates a `MarsRobot` class. This process is called *compiling* the class, and it uses a tool called a compiler. The compiler turns the lines of source code into bytecode that the Java Virtual Machine can run.

The `class` statement in line 1 of Listing 1.1 defines and names the `MarsRobot` class. Everything contained between the opening brace `{` on line 1 and the closing brace `}` on line 18 is part of this class.

The `MarsRobot` class contains three instance variables and two instance methods.

The instance variables are defined in lines 2–4:

```
String status;
int speed;
```

```
    float temperature;
```

The variables are named `status`, `speed`, and `temperature`. Each is used to store a different type of information:

- `status` holds a `String` object—a group of letters, numbers, punctuation, and other characters.
- `speed` holds an `int`, a numeric integer value.
- `temperature` holds a `float`, a floating-point number.

`String` objects are created from the `String` class, which is part of the Java Class Library.

Tip

As you might have noticed from the use of `String` in this program, a class can use an object as an instance variable.

The first instance method in the `MarsRobot` class is defined in lines 6–11:

Click here to view code image

```
    void checkTemperature() {
        if (temperature < -80) {
            status = "returning home";
            speed = 5;
        }
    }
```

Methods are defined in a manner similar to a class. They begin with a statement that names the method, identifies the type of information the method produces, and defines other things.

The `checkTemperature()` method is contained within the opening brace on line 6 of Listing 1.1 and the closing brace on line 11. This method can be called on a `MarsRobot` object to find out its temperature.

This method checks to see whether the object's `temperature` instance variable has a value less than –80. If it does, two other instance variables are changed:

- The `status` variable is changed to the text "returning home," indicating that the temperature is too cold, and the robot is heading back to its base.

- The speed is changed to 5. (Presumably, this is as fast as the robot can travel.)

The second instance method, `showAttributes()`, is defined in lines 13–17:

Click here to view code image

```
void showAttributes() {
    System.out.println("Status: " + status);
    System.out.println("Speed: " + speed);
    System.out.println("Temperature: " + temperature);
}
```

This method calls the method `System.out.println()` to display the values of three instance variables, along with some text explaining what each value represents.

If you haven't saved this file yet, choose File, Save. This command is disabled if the file hasn't been changed since the last time you saved it.

Running the Program

Even if you typed the `MarsRobot` program in Listing 1.1 correctly and compiled it into a class, you can't do anything with it. The class you have created defines what a `MarsRobot` object is like, but it doesn't actually create one of these objects.

There are two ways to put the `MarsRobot` class to use:

- Create a separate Java program that creates an object belonging to that class.
- Add a special class method called `main()` to the `MarsRobot` class so that it can be run as an application. Create an object of that class in that method.

The first option is chosen for this exercise.

Listing 1.2 contains the source code for `MarsApplication`, a Java class that creates a `MarsRobot` object, sets its instance variables, and calls methods. Following the same steps as in the preceding listing, create a new Java file in NetBeans and name it `MarsApplication`.

To begin this second class, follow these steps in NetBeans:

1. Choose File, New File from the menu. The New File dialog opens.

2. In the Categories pane, choose Java.
3. In the File Types pane, choose Empty Java File and click Next. The Empty Java File dialog opens.
4. In the Class Name text field, enter `MarsApplication`. The file you're creating is shown in the Created File field and has the name `MarsApplication.java`.
5. Click Finish.

Enter the code shown in Listing 1.2 into the NetBeans source code editor.

LISTING 1.2 The Full Text of `MarsApplication.java`

Click here to view code image

```
 1: class MarsApplication {
 2:     public static void main(String[] arguments) {
 3:         MarsRobot spirit = new MarsRobot();
 4:         spirit.status = "exploring";
 5:         spirit.speed = 2;
 6:         spirit.temperature = -60;
 7:
 8:         spirit.showAttributes();
 9:         System.out.println("Increasing speed to 3.");
10:         spirit.speed = 3;
11:         spirit.showAttributes();
12:         System.out.println("Changing temperature to -90.");
13:         spirit.temperature = -90;
14:         spirit.showAttributes();
15:         System.out.println("Checking the temperature.");
16:         spirit.checkTemperature();
17:         spirit.showAttributes();
18:     }
19: }
```

When you choose File, Save to save the file, NetBeans automatically compiles it into the `MarsApplication` class, which contains bytecode for the JVM to run.

Tip

If you encounter problems compiling or running any program in this book, you can find a copy of the source file and other related files on the book's official website at www.java21days.com.

After you have compiled the application, run the program by choosing the menu command Run, Run File. The output displayed by the `MarsApplication` class appears in an Output pane in NetBeans, as shown in Figure 1.1.

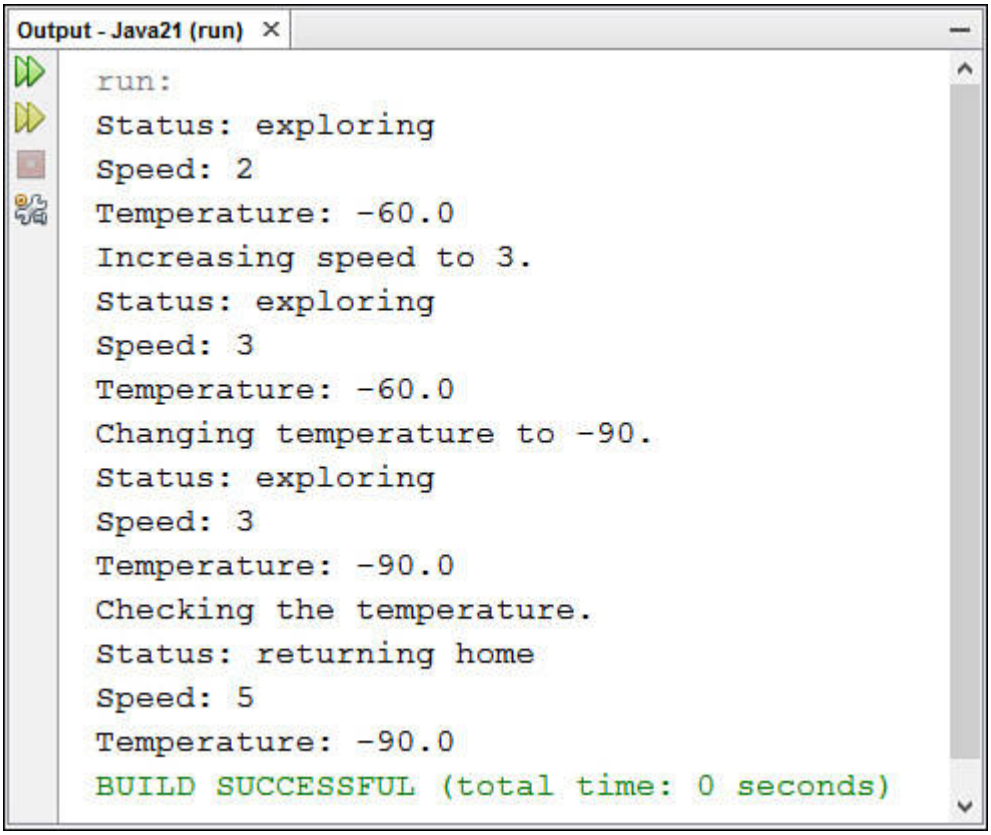


FIGURE 1.1 The output of the `MarsApplication` class.

Using Listing 1.2 as a guide, you can see the following things taking place in the `main()` class method of this application:

- **Line 2**—The `main()` method is created and named. All `main()` methods take this format, as you'll see during Day 5, "Creating Classes and Methods." For now, the most important thing to note is the `static` keyword, which indicates that the method is a class method shared by all `MarsRobot` objects.

- **Line 3**—A new `MarsRobot` object is created using the class as a template. The object is given the name `spirit`.
- **Lines 4–6**—Three instance variables of the `spirit` object are given values: `status` is set to the text "exploring," `speed` is set to 2, and `temperature` is set to –60.
- **Line 8**—On this line and several that follow, the `showAttributes()` method of the `spirit` object is called. This method displays the current values of the instance variables `status`, `speed`, and `temperature`.
- **Line 9**—On this line and others that follow, a call to the `System.out.println()` method displays the text within parentheses to the output device (your monitor).
- **Line 10**—The `speed` instance variable is set to the value 3.
- **Line 13**—The `temperature` instance variable is set to the value –90.
- **Line 16**—The `checkTemperature()` method of the `spirit` object is called. This method checks to see whether the `temperature` instance variable is less than –80. If it is, `status` and `speed` are assigned new values.

Note

If for some reason you can't use NetBeans to write Java programs and must instead use the Java Development Kit, you can find out how to install it in Appendix D, "Using the Java Development Kit," and how to compile and run Java programs with it in Appendix E, "Programming with the Java Development Kit."

Organizing Classes and Class Behavior

Object-oriented programming in Java also requires three more concepts: inheritance, interfaces, and packages. All three are mechanisms for organizing classes and class behavior.

Inheritance

Inheritance, one of the most crucial concepts in object-oriented programming, has a direct impact on how you design and write your own Java classes.

Inheritance is a mechanism that enables one class to inherit the behavior and attributes of another class.

Through inheritance, a class automatically picks up the functionality of an existing class. The new class must only define how it is different from that existing class.

With inheritance, all classes—including those you create and the ones in the Java Class Library—are arranged in a strict hierarchy.

A class that inherits from another class is called a *subclass*. The class that gives the inheritance is called a *superclass*.

A class can have only one superclass, but it can have an unlimited number of subclasses. Subclasses inherit all the attributes and behavior of their superclass.

In practical terms, this means that if the superclass has behavior and attributes that your class needs, you don't have to redefine the behavior or copy that code to have the same behavior and attributes. Your class automatically receives these things from its superclass, the superclass gets them from its superclass, and so on, all the way up the hierarchy. Your class becomes a combination of its own features and all the features of the classes above it in the hierarchy.

The situation is comparable to how you inherited traits from your parents, such as your height, hair color, and love of peanut-butter-and-banana sandwiches. They inherited some of these things from their parents, who inherited from theirs, and backward through time to the Garden of Eden, Big Bang, giant spaghetti monster, or *[insert personal belief here]*.

Figure 1.2 shows how a hierarchy of classes is arranged.

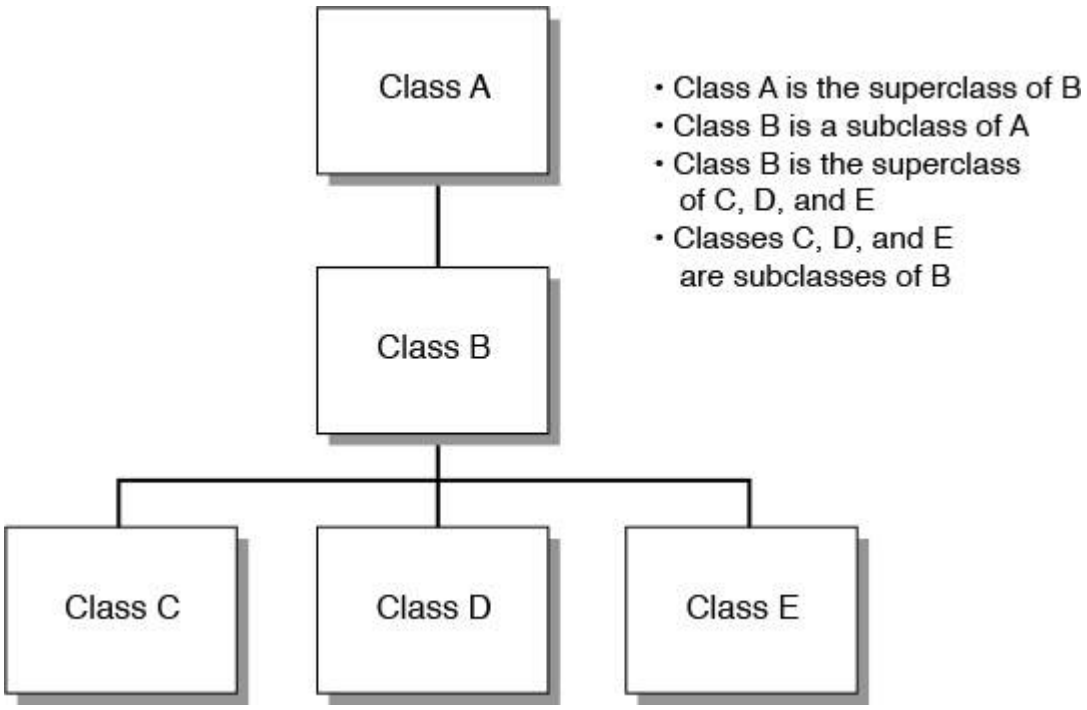


FIGURE 1.2 A class hierarchy.

At the top of the Java class hierarchy is the class `Object`.

All classes inherit from this superclass. `Object` is the most general class in the hierarchy. It defines behavior inherited by all the classes in the Java Class Library.

Each class further down the hierarchy becomes more tailored to a specific purpose. A class hierarchy defines abstract concepts at the top of the hierarchy. Those concepts become more concrete further down the line of subclasses.

Often when you create a new class in Java, you want all the functionality of an existing class except for some additions or modifications of your own creation. For example, you might want a new version of `CommandButton` that makes a sound when clicked.

To receive all the `CommandButton` functionality without doing any work to re-create it, you can define your new class as a subclass of `CommandButton`.

Because of inheritance, your class automatically inherits behavior and attributes defined in `CommandButton` as well as the behavior and attributes defined in the superclasses of `CommandButton`. All you have to worry about are the things that make your new class different from

`CommandButton` itself. Subclassing is the mechanism for defining new classes as the differences between those classes and their superclass.

Subclassing is the creation of a new class that inherits from an existing class. The only task in the subclass is to indicate the differences in behavior and attributes between the subclass and its superclass.

If your class defines entirely new behavior and isn't a subclass of another class, you can inherit directly from the `Object` class.

If you create a class that doesn't indicate a superclass, Java assumes that the new class inherits directly from `Object`. The `MarsRobot` class you created earlier today did not specify a superclass, so it's a subclass of `Object`.

Creating a Class Hierarchy

If you're creating a large set of classes, it makes sense for your classes to inherit from the existing class hierarchy and to make up a hierarchy themselves. This gives your classes several advantages:

- Functionality common to multiple classes can be put into a superclass, which enables it to be used repeatedly in all classes below it in the hierarchy.
- Changes to a superclass automatically are reflected in all its subclasses, their subclasses, and so on. There is no need to change or recompile any of the lower classes; they receive the new information through inheritance.

For example, imagine that you have created a Java class to implement all the features of an exploratory robot. (This shouldn't take much imagination.)

The `MarsRobot` class is completed and works successfully. Your boss at NASA asks you to create a Java class called `MercuryRobot`.

These two kinds of robots have similar features. Both are research robots that work in hostile environments and conduct research. Both keep track of their current temperature and speed.

Your first impulse might be to open the `MarsRobot.java` source file, copy it into a new source file called `MercuryRobot.java`, and then make the necessary changes for the new robot to do its job.

A better plan is to figure out the common functionality of `MercuryRobot` and `MarsRobot` and organize it into a more general class hierarchy. This might be a lot of work just for the classes `MarsRobot` and `MercuryRobot`, but what if you also want to add `MoonRobot`, `UnderseaRobot`, and `DesertRobot`? Factoring common behavior into one or more reusable superclasses significantly reduces the overall amount of work you must do.

To design a class hierarchy that might serve this purpose, start at the top with the class `Object`, the pinnacle of all Java classes.

The most general class to which these robots belong might be called `Robot`. A robot, generally, could be defined as a self-controlled exploration device. In the `Robot` class, you define only the behavior that qualifies something to be a device, to be self-controlled, and to be designed for exploration.

There could be two classes below `Robot`: `WalkingRobot` and `DrivingRobot`. The obvious thing that differentiates these classes is that one travels by foot and the other by wheel. The behavior of walking robots might include bending over to pick up something, ducking, running, and the like. Driving robots would behave differently. Figure 1.3 shows what you have so far.

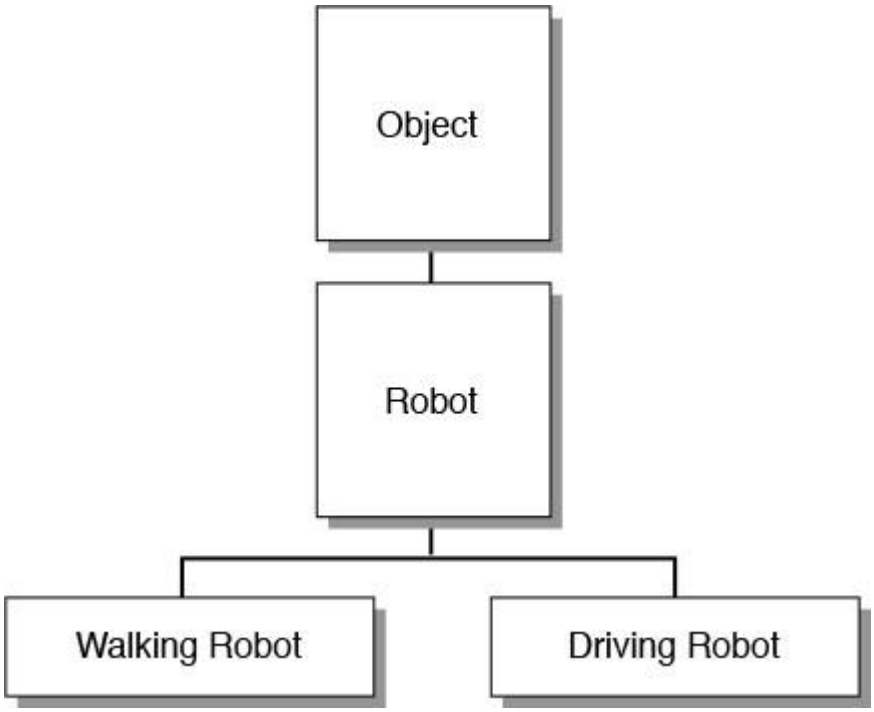


FIGURE 1.3 The basic `Robot` hierarchy.

Now, the hierarchy can become even more specific.

With `WalkingRobot`, you might have several classes: `ScienceRobot`, `GuardRobot`, `SearchRobot`, and so on. As an alternative, you could factor out still more functionality and have intermediate classes for `TwoLegged` and `FourLegged` robots, with different behaviors for each (see Figure 1.4).

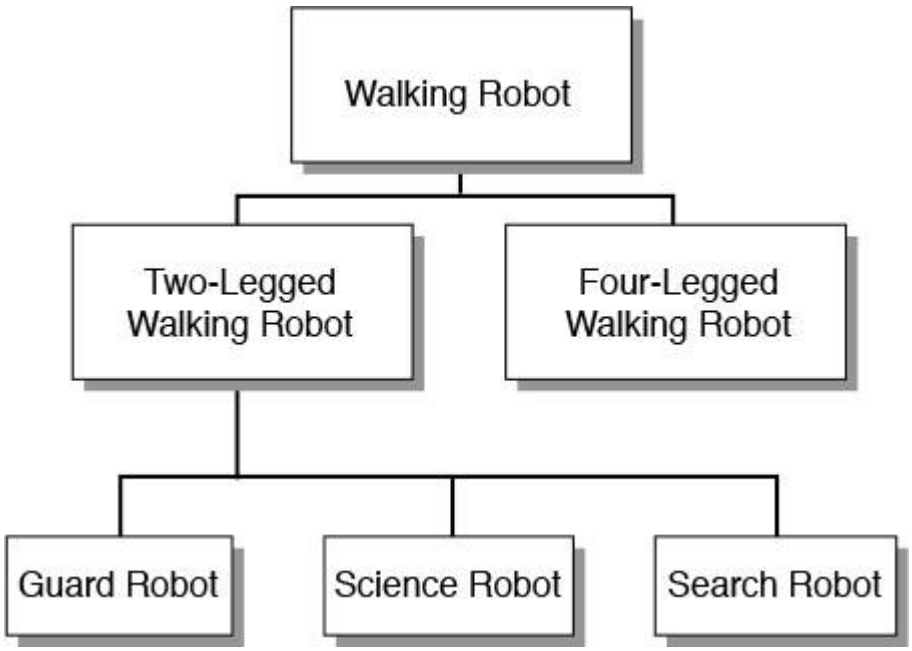


FIGURE 1.4 Two-legged and four-legged walking robots.

Finally, the hierarchy is done, and you have a place for `MarsRobot`. It can be a subclass of `ScienceRobot`, which is a subclass of `WalkingRobot`, which is a subclass of `Robot`, which is a subclass of `Object`.

Where do attributes such as status, temperature, and speed come in? At the place they fit into the class hierarchy most naturally. Because all robots need to keep track of the temperature of their environment, it makes sense to define `temperature` as an instance variable in `Robot`. All subclasses would have that instance variable as well. Remember that you need to define a behavior or attribute only once in the hierarchy, and it is inherited automatically by each subclass.

Note

Designing an effective class hierarchy involves a lot of planning and revision. As you attempt to put attributes and behavior into a hierarchy, you're likely to find reasons to move some classes to different spots in the hierarchy. The goal is to reduce the number of repetitive features (and redundant code) needed.

Inheritance in Action

Inheritance in Java works much more simply than it does in the real world. No wills or courts are required when inheriting from a parent.

When you create a new object, Java keeps track of each variable defined for that object and each variable defined for each superclass of the object. In this way, all the classes combine to form a template for the current object, and each object fills in the information appropriate to its situation.

Methods operate similarly. A new object has access to all method names of its class and superclass. This is determined dynamically when a method is used in a running program. If you call a method of a particular object, the Java virtual machine first checks the object's class for that method. If the method isn't found, the virtual machine looks for it in the superclass of that class, and so on, until the method definition is found. This is illustrated in Figure 1.5.

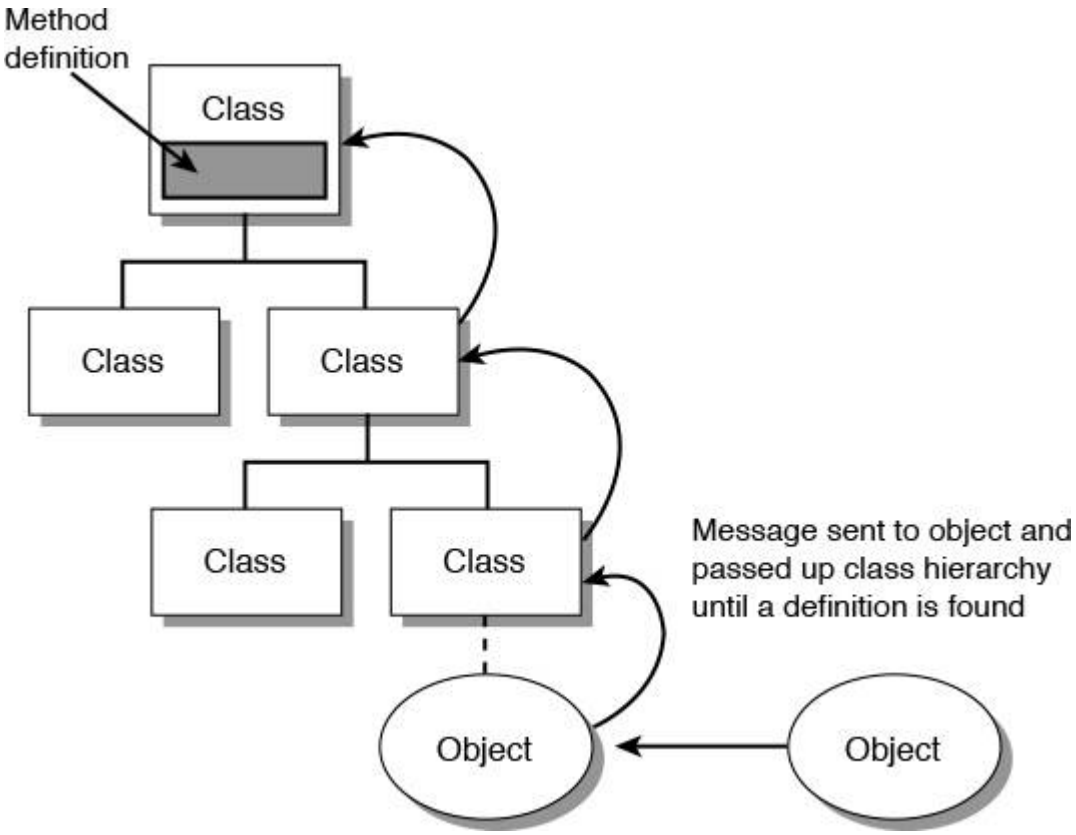


FIGURE 1.5 How methods are located in a class hierarchy.

Things get complicated when a subclass defines a method that matches a method defined in a superclass in name and other aspects. In this case, the method definition found first (starting at the bottom of the hierarchy and working upward) is the one that is used.

Because of this, you can create a method in a subclass that prevents a method in a superclass from being used. To do this, you give the method the same name, return type, and arguments as the method in the superclass. This procedure, shown in Figure 1.6, is called *overriding*.

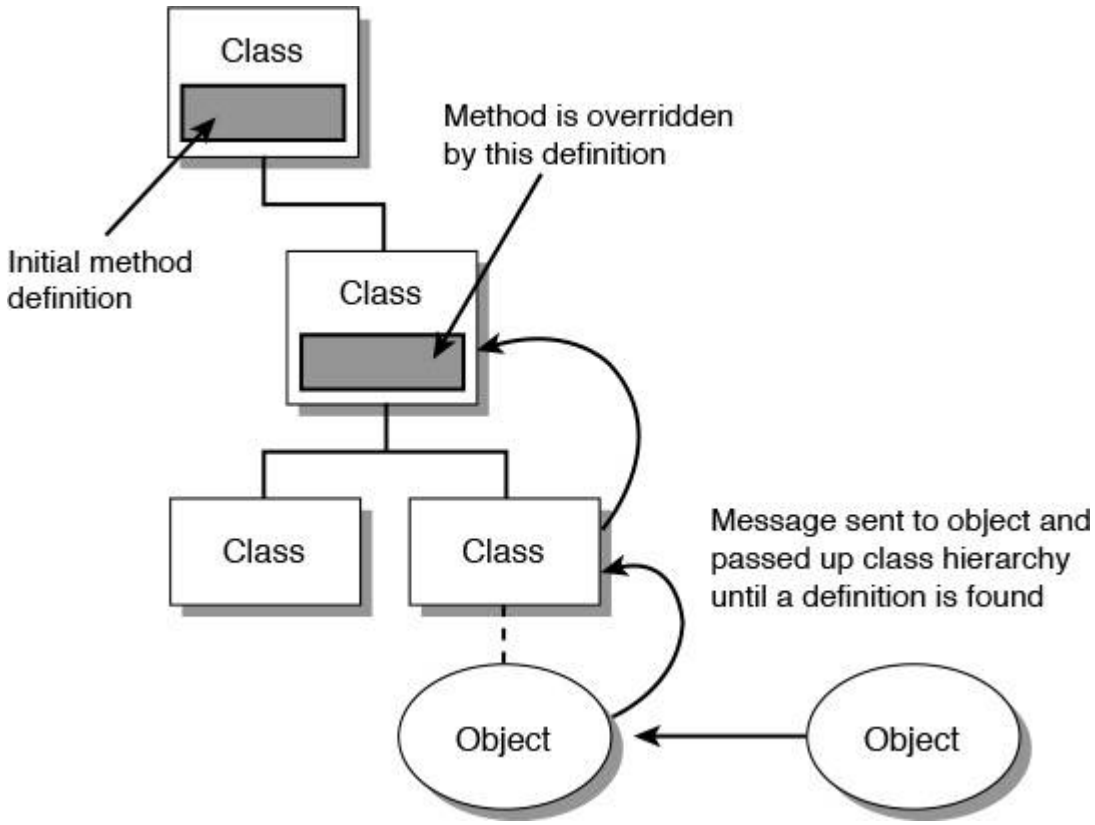


FIGURE 1.6 Overriding methods.

Note

Java's form of inheritance is called single inheritance because each Java class can have only one superclass, although any given superclass can have multiple subclasses.

In other object-oriented programming languages such as C++, classes can have more than one superclass, and they inherit combined variables and methods from all those superclasses. This is called multiple inheritance. Java makes inheritance simpler by allowing only single inheritance.

Interfaces

Single inheritance makes the relationship between classes and the functionality they implement easier to understand and design. However, it also can be restrictive, especially when you have similar behavior that needs to be duplicated across different branches of a class hierarchy. Java solves the problem of shared behavior by using interfaces.

An *interface* is a collection of methods that indicate a class has some behavior in addition to what it inherits from its superclasses. The methods included in an interface do not define this behavior; that task is left for the classes that implement the interface.

For example, the `Comparable` interface contains a method that compares two objects of the same class to see which one should appear first in a sorted list. Any class that implements this interface shows other objects that it knows how to determine the sorting order for objects of that class. This behavior would be unavailable to the class without the interface.

You'll learn about interfaces during Day 6, "Packages, Interfaces, and Other Class Features."

Packages

Packages in Java are a way to group related classes and interfaces. *Packages* enable groups of classes to be referenced more easily in other classes. They also eliminate potential naming conflicts among classes.

Classes in Java can be referred to by a short name such as `Object` or a full name such as `java.lang.Object`.

By default, your Java classes can refer to the classes in the `java.lang` package using only short names. The `java.lang` package provides basic language features such as string handling and mathematical operations. To use classes from any other package, you must refer to them explicitly using their full package name or use an `import` command to import the package in your source code file.

Because the `Color` class is contained in the `java.awt` package, you normally refer to it in your programs with the notation `java.awt.Color`.

If the entire `java.awt` package has been imported using `import`, the class can be referred to as `Color`.

The package for a class is determined by the `package` statement. Many of the classes you create in this book are put in the com.java24hours package, like so:

```
package com.java24hours;
```

This statement must be the first line of the program. When it is omitted, as it was in the `MarsRobot` and `MarsApplication` programs you created

today, the class belongs to an unnamed package called the default package.

Summary

If today was your first exposure to object-oriented programming, it probably seemed theoretical and a bit overwhelming.

Because your brain has been stuffed with object-oriented programming concepts and terminology for the first time, you might be worried that no room is left for the Java lessons of the remaining 20 days.

Don't panic. Keep calm and carry on.

At this point, you should have a basic understanding of classes, objects, attributes, and behavior. You also should be familiar with instance variables and methods. You'll use these right away tomorrow.

The other aspects of object-oriented programming, such as inheritance and packages, will be covered in more detail in upcoming days.

You'll work with object-oriented programming in every remaining day of the book. There's no other way to create programs in Java.

By the time you finish the first week, you'll have working experience with objects, classes, inheritance, and all other aspects of the methodology.

Q&A

Q Methods are functions defined inside classes. If they look like functions and act like functions, why aren't they called functions?

A Some object-oriented programming languages do call them functions. (C++ calls them member functions.) Other object-oriented languages differentiate between functions inside and outside the body of a class or object because in those languages the use of the separate terms is important to understanding how each function works. Because the difference is relevant in other languages and because the term *method* now is in common use in object-oriented terminology, Java uses the term as well.

Q What's the distinction between instance variables and methods and their counterparts, class variables and methods?

A Almost everything you do in a Java program involves instances (also called objects) rather than classes. However, some behavior and

attributes make more sense if stored in the class itself rather than in the object.

For example, the `Math` class in the `java.lang` package includes a class variable called `PI` that holds the approximate value of pi. This value does not change, so there's no reason why different objects of that class would need their own individual copy of the `PI` variable. On the other hand, every `String` object contains a method called `length()` that reveals the number of characters in that `String`. This value can be different for each object of that class, so it must be an instance method.

Class variables occupy memory until a Java program is finished running, so they should be used with care. If a class variable references an object, that object will remain in memory as well. This is a common problem causing a program to take up too much memory and run slowly.

Q When a Java class imports an entire package, does it increase the compiled size of that class?

A No. The use of the term "import" is a bit misleading. The `import` keyword does not add the bytecode of one class or one package to the class you are creating. Instead, it makes it easier to refer to classes within another class.

The sole purpose of importing is to shorten the class names when they're used in Java statements. It would be cumbersome to always have to refer to full class names such as `javax.swing.JButton` and `java.awt.Graphics` in your code instead of calling them `JButton` and `Graphics`.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What is another word for a class?

   A. Object

   B. Template

C. Instance

2. When you create a subclass, what must you define about that class?

   A. Nothing. Everything is defined already.

   B. Things that are different from its superclass

   C. Everything about the class

3. What does an instance method of a class represent?

   A. The attributes of that class

   B. The behavior of that class

   C. The behavior of an object created from that class

Answers

1. B. A class is an abstract template used to create objects that are similar to each other.
2. B. You define how the subclass is different from its superclass. The things that are similar are already defined for you because of inheritance. Answer A is technically correct, but if everything in the subclass is identical to the superclass, there's no reason to create the subclass.
3. C. Instance methods refer to a specific object's behavior. Class methods refer to the behavior of all objects belonging to that class.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material.

Which of the following statements is true?

A. All objects created from the same class must be identical.

B. All objects created from the same class can have different attributes than each other.

C. An object inherits attributes and behavior from the class used to create it.

D. A class inherits attributes and behavior from its subclass.

The answer is available on the book's website at www.java21days.com. Visit the Day 1 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. In the `main()` method of the `MarsRobot` class, create a second `MarsRobot` robot named `opportunity`, set up its instance variables, and display them.
2. Create an inheritance hierarchy for the pieces of a chess set. Decide where the instance variables `color`, `startingPosition`, `forwardMovement`, and `sideMovement` should be defined in the hierarchy.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 2. The ABCs of Programming

A Java program is made up of classes and objects, which, in turn, are made up of methods and variables. Methods are made up of statements and expressions, which are made up of operators.

At this point, you might be worried that Java is like a set of Russian nesting matryoshka dolls. Each doll has a smaller doll inside it, as intricate and detailed as its larger companion, until you reach the smallest one.

Today's lesson clears away the big dolls to reveal the smallest elements of Java programming. You will set aside classes, objects, and methods for a day and examine the basic things you can do in a single line of Java code.

The following subjects are covered:

- Statements and expressions
- Variables and primitive data types
- Constants
- Comments
- Literals
- Arithmetic
- Comparisons
- Logical operators

Statements and Expressions

All the tasks you want to accomplish in a Java program can be broken into a series of statements. In a programming language, a *statement* is a simple command that causes something to happen.

Statements represent a single action taken in a Java program. Here are three simple Java statements:

Click here to view code image

```
int weight = 225;
System.out.println("Free the bound periodicals!");
song.duration = 230;
```

Some statements can convey a value, such as when two numbers are added or two variables are compared to find out if they are equal.

A statement that produces a value is called an *expression*. The value can be stored for later use in the program, used immediately in another statement, or disregarded. The value produced by a statement is called its *return value*.

Some expressions produce a numeric return value, as when two numbers are added or multiplied. Others produce a Boolean value—either `true` or `false`—or even can produce a Java object. They are discussed later today.

Although many Java programs contain one statement per line, this is a formatting decision that does not determine where one statement ends and another one begins. Each statement in Java is terminated with a semicolon character `;`. A programmer can put more than one statement on a line and it will compile successfully, as in the following example:

Click here to view code image

```
spirit.speed = 2; spirit.temperature = -60;
```

To make your program more readable to other programmers (and yourself), you should follow the convention of putting only one statement on each line.

Statements in Java are grouped using an opening brace `{` and a closing brace `}`. A group of statements organized between these characters is called a block (or block statement). You learn more about them during Day 4, "Lists, Logic, and Loops."

Variables and Data Types

In the MarsRobot application created during Day 1, "Getting Started with Java," you used variables to keep track of information. A variable is a place where information can be stored while a program is running. The value can be changed at any point in the program—hence the name.

To create a variable, you must give it a name and identify the type of information it will store. You also can give a variable an initial value at the same time you create it.

Java has three kinds of variables: instance variables, class variables, and local variables.

Instance variables, as you learned yesterday, define an object's attributes.

Class variables define the attributes of an entire class of objects and apply to all instances of it.

Local variables are used inside method definitions or even smaller blocks of statements within a method. You can use them only while the method or block is being executed by the Java Virtual Machine. They cease to exist afterward.

Although all three kinds of variables are created in much the same way, class and instance variables are used in a different manner than local variables. You learn about local variables today and explore instance and class variables during Day 3, "Working with Objects."

Creating Variables

Before you can use a variable in a Java program, you must create the variable by declaring its name and the type of information it will store. The type of information is listed first, followed by the name of the variable. The following all are examples of variable declarations:

```
int loanLength;
String message;
boolean gameOver;
```

In these examples, the `int` type represents integers, `String` is an object that holds text, and `boolean` is used for Boolean true/false values.

Local variables can be declared at any place inside a method, like any other Java statement, but they must be declared before they can be used.

In the following example, three variables are declared at the top of a program's `main()` method:

Click here to view code image

```
public static void main(String[] arguments) {
    int total;
    String reportTitle;
    boolean active;
}
```

If you are creating several variables of the same type, you can declare all of them in the same statement by separating the variable names with commas. The following statement creates three `String` variables named `street`, `city`, and `state`:

```
String street, city, state;
```

Variables can be assigned a value when they are created by using an equal sign (=) followed by the value. The following statements create new

variables and give them initial values:

Click here to view code image

```
String zipCode = "02134";

int box = 350;
boolean pbs = true;
String name = "Zoom", city = "Boston", state = "MA";
```

As the last statement demonstrates, you can assign values to multiple variables of the same type by using commas to separate them.

You must give values to local variables before you use them in a program, or the program won't compile successfully. For this reason, it is good practice to give initial values to all local variables.

Instance and class variable definitions are given an initial value depending on the type of information they hold, as in the following:

- Numeric variables: `0`
- Characters: `"\0"`
- Booleans: `false`
- Objects: `null`

Naming Variables

Variable names in Java must start with a letter, an underscore character `_`, or a dollar sign `$`.

Variable names cannot start with a number. After the first character, variable names can include any combination of letters, numbers, underscore characters, or dollar signs.

Note

In addition, the Java language uses the Unicode character set, which includes thousands of character sets to represent international alphabets. Accented characters and other symbols can be used in variable names as long as they have a Unicode character number.

When naming a variable and using it in a program, it's important to remember that Java is case-sensitive—the capitalization of letters must be consistent.

Because of this, a program can have a variable named `X` and another named `x` (so `Rose` is not `rose` is not `ROSE`).

In programs in this book and elsewhere, Java variables are given meaningful names that include several joined words. To make it easier to spot the words, the following general rules are used:

- The first letter of the variable name is lowercase.
- Each successive word in the variable name begins with a capital letter.
- All other letters are lowercase.

The following variable declarations follow these naming rules:

```
Button loadFile;
int localAreaCode;
boolean quitGame;
```

Although dollar signs and underscores are permitted in variable names, you should avoid using either of them except in one situation: When a variable's entire name is capitalized, each word is separated by an underscore. Here's an example:

```
static int DAYS_IN_WEEK = 7;
```

You will see why a variable name might be capitalized like this later today.

Dollar signs never should be used in variable names, even though they're permitted. The official documentation for Java always has discouraged their use, so programmers follow this convention.

Variable Types

In addition to a name, a variable declaration must include the data type of information being stored. The type can be any of the following:

- One of the primitive data types, such as `int` or `boolean`
- The name of a class or interface
- An array

You learn how to declare and use array variables on Day 4. Today's lesson focuses on the other variable types.

Data Types

Java has eight basic data types that store integers, floating-point numbers, characters, and Boolean values. These often are called *primitive types* because they are built-in parts of the language rather than objects, which makes them easier to create and use. These data types have the same size and characteristics no matter what operating system and platform you're on, unlike some data types in other programming languages.

You can use four data types to store integers. Which one you use depends on the integer's size, as shown in Table 2.1.

Type	Size	Values That Can Be Stored
`byte`	8 bits	–128 to 127
`short`	16 bits	–32,768 to 32,767
`int`	32 bits	–2,147,483,648 to 2,147,483,647
`long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

TABLE 2.1 Integer Types

All these types are *signed*, which means that they can hold either positive or negative numbers. The type used for a variable depends on the range of values it might need to hold. None of these integer variables can reliably store a value that is too large or too small for its designated variable type, so take care when designating the type.

Another type of number that can be stored is a floating-point number, which has the type `float` or `double`. *Floating-point* numbers are numbers with a decimal point. The `float` type can handle any number from `1.4E-45` to `3.4E+38`, while the `double` type can be used for more precise numbers ranging from `4.9E-324` to `1.7E+308`. Because `double` has more precision, that type generally is preferred.

The `char` type is used for individual characters, such as letters, numbers, punctuation, and other symbols.

The last of the eight primitive data types is `boolean`. As you have learned, this data type holds either `true` or `false`.

All these variable types appear in lowercase, and you must use them as such in programs. Some classes have the same names as these data types, but with different capitalization, such as `Boolean` and `Double` These are created

and referenced differently in a Java program, so you can't use them interchangeably in most circumstances. Tomorrow you will see how to use these special classes.

Note

There's actually a ninth primitive data type in Java, `void`, which represents nothing. It's used in methods to indicate that they do not return a value.

Class Types

In addition to the primitive data types, a variable can have a class as its type, as in the following examples:

```
String lastName = "Hopper";
Color hair;
MarsRobot robbie;
```

When a variable has a class as its type, the variable refers to an object of that class or one of its subclasses.

The last statement in the preceding list creates a variable named `robbie` that is reserved for a `MarsRobot` object. You learn more tomorrow about how to associate objects with variables.

Assigning Values to Variables

After a variable has been declared, a value can be assigned to it with the assignment operator, which is an equal sign =. The following are examples of assignment statements:

```
idCode = 8675309;

accountOverdrawn = false;
```

Constants

Variables are useful when you need to store information that can be changed as a program runs.

If the value never should change during a program's runtime, you can use a type of variable called a constant. A *constant* is a variable with a value that

never changes. (This might seem like an oxymoron, given the meaning of the word “variable.”)

Constants are useful in defining shared values for the use of all methods of an object. In Java, you can create constants for all kinds of variables: instance, class, and local.

To declare a constant, use the `final` keyword before the variable declaration and include an initial value for that variable, as in the following:

```
final double PI = 3.141592;
final boolean DEBUG = false;
final int PENALTY = 25;
```

Constants can be handy for naming various states of an object and then testing for those states. Suppose you have a program that takes directional input from the numeric keypad on the keyboard—press 8 to go up, 4 to go left, 6 to go right, and 2 to go down. You can define those values as constant integers:

```
final int LEFT = 4;
final int RIGHT = 6;
final int UP = 8;
final int DOWN = 2;
```

Constants often make a program easier to understand. To illustrate this point, consider which of the following two statements is more informative as to its function:

```
guide.direction = 4;
guide.direction = LEFT;
```

Note

In the preceding statements, the names of the constants such as `DEBUG` and `LEFT` are capitalized. This is a convention adopted by Java programmers to make it clear that the variable is a constant. Java does not require that constants be capitalized in this manner, but it’s a good practice to adopt.

When a constant’s variable name is more than one word, putting it in all caps would make the words run together confusingly, as in ESCAPECODE. Separate the words with an underscore character _, like this:

```
final int ESCAPE_CODE = 27;
```

Today's first project is a Java application that creates several variables, assigns them initial values, and displays two of them as output. Run NetBeans and create a new Java program by undertaking these steps, which have one difference from the procedure in Day 1:

1. Choose the menu command File, New File. The New File dialog box opens.
2. In the Categories pane, choose Java.
3. In the File Types pane, choose Empty Java File and click Next. The Empty Java File dialog box opens.
4. In the Class Name text field, enter `Variables`, which will give the source code file the name `Variables.java`.
5. Here's the different step: In the Package Name text field, enter `com.java21days`.
6. Click Finish.

On this project, you specify a class name and a package name. Packages are a way to organize related Java programs together. They serve a similar purpose to file folders in a file system. Enter the code shown in Listing 2.1 into the source code editor.

LISTING 2.1 The Full Text of `Variables.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: public class Variables {
 4:
 5:     public static void main(String[] arguments) {
 6:         final char UP = 'U';
 7:         byte initialLevel = 12;
 8:         short location = 13250;
 9:         int score = 3500100;
10:         boolean newGame = true;
11:
12:         System.out.println("Level: " + initialLevel);
13:         System.out.println("Up: " + UP);
14:     }
15: }
```

Save the file by choosing File, Save. NetBeans automatically compiles the application if it contains no errors. Run the program by choosing Run, Run File. This program produces the output shown in Figure 2.1.

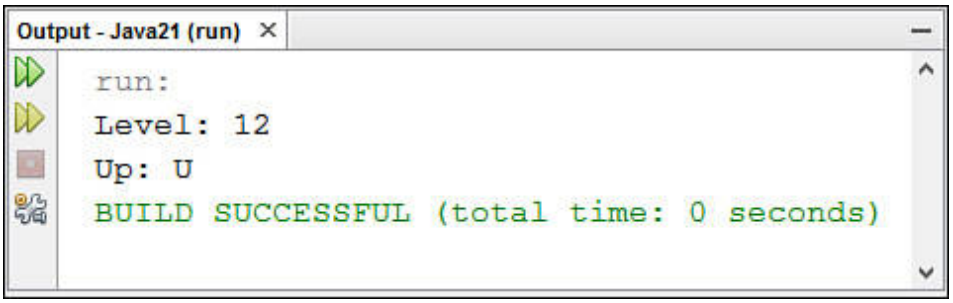


FIGURE 2.1 Creating and displaying variable values.

The package name of the class is established by the `package` statement, which must be the first line of a Java program when it is used:

```
package com.java21days;
```

This class uses four local variables and one constant, making use of `System.out.println()` in lines 12–13 to produce output.

`System.out.println()` is a method called to display strings and other information to the standard output device, which usually is the monitor.

This method takes a single argument within its parentheses: a string. To present more than one variable or literal as the argument to `println()`, the `+` operator combines the elements into a single string.

Java also has a `System.out.print()` method that displays a string without terminating it with a newline character. You can call `print()` instead of `println()` to display several strings on the same line.

Comments

One of the most effective ways to improve a program's readability is to use comments. These are text included in a program that explains what's going on in the code. The Java compiler ignores comments when preparing a bytecode version of a Java source file that can be run as a class, so there's no penalty for using them.

You can use three kinds of comments in Java programs.

A single-line comment is preceded by two slash characters `//`. Everything from the slashes to the end of the line is considered a comment and is disregarded by the compiler, as in the following statement:

Click here to view code image

```
int creditHours = 3; // set up credit hours for course
```

Everything from the slashes onward is ignored. As far as the compiler is concerned, the preceding line is the same as this:

```
int creditHours = 3;
```

A multiline comment begins with /* and ends with */. Everything between these two delimiters is considered a comment, even over multiple lines, as in the following code:

Click here to view code image

```
/* This program occasionally deletes all files on
your hard drive and renders it unusable
forever when you click the Save button. */
```

A Javadoc comment begins with /** and ends with */. Everything between these delimiters is considered to be official documentation on how the class and its methods work.

Javadoc comments are designed to be read by utilities such as javadoc, a command-line tool that's part of the Java Development Kit (JDK). This tool uses official comments to create a set of web pages that document the functionality of a Java class, show its place in relation to its superclass and subclasses, and describe each of its methods.

Tip

All the official documentation on each class in the Java Class Library is generated from Javadoc comments. You can view current Java documentation at http://docs.oracle.com/javase/8/docs/api.

You learn how to use the javadoc tool in Appendix E, "Programming with the Java Development Kit."

Literals

In addition to variables, you can work with values as literals in a Java statement. A *literal* is any number, text, or other information that directly represents a value.

The following assignment statement uses a literal:

```
int year = 2016;
```

The literal 2016 represents the integer value 2016. Numbers, characters, and strings are all examples of literals. Java has types of literals that represent different kinds of numbers, characters, strings, and Boolean values.

Number Literals

Java has several integer literals. The number 4, for example, is an integer literal of the `int` variable type. It also can be assigned to `byte` and `short` variables, because the number is small enough to fit into those integer types. An integer literal larger than an `int` can hold automatically is considered to be of the type `long`. You also can indicate that a literal should be a `long` integer by adding the letter L to the number (either in upper- or lowercase). Here's an example:

```
pennyTotal = pennyTotal + 4L;
```

This statement adds the value 4, formatted as a `long`, to the current value of the `pennyTotal` variable.

To represent a negative number as a literal, precede it with a minus sign (–), as in –45.

Floating-point literals use a period character. for the decimal point, as you would expect. The following statement uses a literal to set up a `double` variable:

```
double gpa = 3.55;
```

All floating-point literals are considered to be of the `double` variable type instead of `float`. To specify a literal of `float`, add the letter F to the literal (upper- or lowercase), as in the following example:

```
float piValue = 3.1415927F;
```

You can use exponents in floating-point literals by using the letter e or E followed by the exponent, which can be a negative number. The following statements use exponential notation:

```
double x = 12e22;

double y = 19E-95;
```

A large integer literal can include an underscore character _ to make it more readable to humans. The underscore serves the same purpose as a comma in a large number, making its value more apparent. Consider these two examples, one of which uses underscores:

```
int jackpot = 3500000;

int jackpot = 3_500_000;
```

Both examples equal 3,500,000, which is easier to see in the second statement. The Java compiler ignores the underscores.

Java also supports numeric literals that use binary, octal, and hexadecimal numbering.

Binary numbers are a base-2 numbering system in which only the values 0 and 1 are used. Values made up of 1s and 0s are the simplest form for a computer and are a fundamental part of computing. Counting up from 0, binary values are 0, 1, 10, 11, 100, 111, and so on. Each digit in the number is called a bit. The combination of eight numbers is a byte. A binary literal is specified by preceding it with 0b, as in 0b101 for 101 (5 in decimal) and 0b01111111 (127).

Octal numbers are a base-8 numbering system, which means that they can represent only the values 0 through 7 as a single digit. The eighth number in octal is 10. Octal literals begin with a 0, so 010 is the decimal value 8, 012 is 9, and 020 is 16.

Hexadecimal is a base-16 numbering system that can represent 16 numbers as a single digit. The letters A through F represent the last six digits, so the first 16 numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Hexademical literals begin with 0x, as on 0x12 (decimal 18) and 0xFF (255).

The octal and hexadecimal systems are better suited for certain tasks in programming than the normal decimal system. If you ever have edited a web page to set its background color, you could have used hexadecimal numbers for green (0x001100), blue (0x000011), or butterscotch (0xFFCC99).

Boolean Literals

The Boolean literals `true` and `false` are the only two values you can use when assigning a value to a `boolean` variable type or using a Boolean in a statement.

The following statement sets a `boolean` variable:

```
boolean chosen = true;
```

Caution

If you have programmed in other languages, you might expect that a value of 1 is equivalent to true and 0 is equivalent to false. This isn't the case in Java; you must use the values `true` and `false` to represent Boolean values.

Note that the literal `true` does not have quotation marks around it. If it did, the Java compiler would assume that it is a string of characters.

Character Literals

Character literals are expressed by a single character surrounded by single quotation marks, such as 'a', '#', and '3'. You might be familiar with the ASCII character set, which includes 128 characters, including letters, numerals, punctuation, and other characters useful in computing. Java supports ASCII along with thousands of additional characters through the 16-bit Unicode standard.

Some character literals represent characters that are not readily printable or accessible from a keyboard. Table 2.2 lists the codes that can represent these special characters as well as characters from the Unicode character set.

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
d	Octal
\x*d*	Hexadecimal
\u*d*	Unicode character

TABLE 2.2 Character Escape Codes

In Table 2.2, the letter *d* in the octal, hex, and Unicode escape codes represents a number or a hexadecimal digit (a through f or A through F).

String Literals

The final literal that you can use in a Java program represents strings of characters. A string in Java is an object rather than a primitive data type. Strings are not stored in arrays as they are in languages such as C.

Because string objects are real objects in Java, methods are available to combine strings, modify strings, and determine whether two strings have the same value.

String literals consist of a series of characters inside double quotation marks, as in the following statements:

Click here to view code image

```
String quitMsg = "Are you sure you want to quit?";

String password = "drowssap";
```

Strings can include the character escape codes listed in Table 2.2, as shown here:

Click here to view code image

```
String example = "Socrates asked, \"Hemlock is poison?\"";

System.out.println("Sincerely,\nMillard Fillmore\n");

String title = "Sams Teach Yourself Node in the John\u2122";
```

In the last example, the Unicode code sequence `\u2122` produces a ™ symbol on systems that have been configured to support Unicode.

Caution

Although Java supports the transmission of Unicode characters, a computer also must support it for the characters to be displayed when the program is run. Unicode support provides a way to encode its characters for systems that support the standard. Java supports the display of any Unicode character that can be represented by a host font.

For more information about Unicode, visit the Unicode Consortium website at www.unicode.org.

Although string literals are used in a manner similar to other literals in a program, they are handled differently behind the scenes.

With a string literal, Java stores that value as a `String` object. You don't have to explicitly create a new object, as you must when working with other objects, so they are as easy to work with as primitive data types. Strings are unusual in this respect—none of the basic types are stored as an object when used. You'll learn more about strings and the `String` class later today.

Expressions and Operators

An *expression* is a statement that can convey a value. Some of the most common expressions are mathematical, such as in the following examples:

```
int x = 3;
int y = x;
int z = x * y;
```

All three of these statements can be considered expressions; they convey values that can be assigned to variables. The first assigns the literal 3 to the variable `x`. The second assigns the value of the variable `x` to the variable `y`. In the third expression, the multiplication operator * is used to multiply the `x` and `y` integers, and the result is stored in the `z` integer.

Expressions can be any combination of variables, literals, and operators. They also can be method calls because methods send back a value to the object or class that called the method.

The value conveyed by an expression is called a *return value*. This value can be assigned to a variable and used in many other ways in your Java programs.

Most of the expressions in Java use operators such as *. *Operators* are special symbols used for mathematical functions, assignment statements, and logical comparisons.

Arithmetic

Five operators are used to accomplish basic arithmetic in Java, as shown in Table 2.3.

Operator	Meaning	Example
+	Addition	`3 + 4`
-	Subtraction	`5 - 7`
*	Multiplication	`5 * 5`
/	Division	`14 / 7`
%	Modulus	`20 % 7`

TABLE 2.3 Arithmetic Operators

Each operator takes two operands, one on each side of the operator. The subtraction operator also can be used to negate a single operand, which is equivalent to multiplying that operand by –1.

One thing to be mindful of when performing division is the type of numbers being used. If you store a division operation in an integer, the result is truncated to the next-lower whole number, because the `int` data type can't handle floating-point numbers.

For example, the expression `31 / 9` results in 3 if stored as an integer.

Modulus division, which uses the `%` operator, produces the remainder of a division operation. The expression `31 % 9` results in 4 because 31 divided by 9, with the whole number result of 3, leaves a remainder of 4.

Note that many arithmetic operations involving integers produce an `int` regardless of the original type of the operands. If you're working with other

numbers, such as floating-point numbers or `long` integers, you should make sure that the operands have the same type you're trying to end up with.

The next project is a Java class that demonstrates how to perform simple arithmetic in the language. Create a new empty Java file in NetBeans called `Weather` in the `com.java21days` package and enter the code shown in Listing 2.2 into the source code editor. Save the file with the menu command File, Save when you're done.

LISTING 2.2 The Full Text of `Weather.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: public class Weather {
 4:     public static void main(String[] arguments) {
 5:         float fah = 86;
 6:         System.out.println(fah + " degrees Fahrenheit is
...");
 7:         // To convert Fahrenheit into Celsius
 8:         // begin by subtracting 32
 9:         fah = fah - 32;
10:         // Divide the answer by 9
11:         fah = fah / 9;
12:         // Multiply that answer by 5
13:         fah = fah * 5;
14:         System.out.println(fah + " degrees Celsius\n");
15:
16:         float cel = 33;
17:         System.out.println(cel + " degrees Celsius is ...");
18:         // To convert Celsius into Fahrenheit
19:         // begin by multiplying by 9
20:         cel = cel * 9;
21:         // Divide the answer by 5
22:         cel = cel / 5;
23:         // Add 32 to the answer
24:         cel = cel + 32;
25:         System.out.println(cel + " degrees Fahrenheit");
26:     }
27: }
```

Run the program by selecting Run, Run File. It produces the output shown in Figure 2.2.

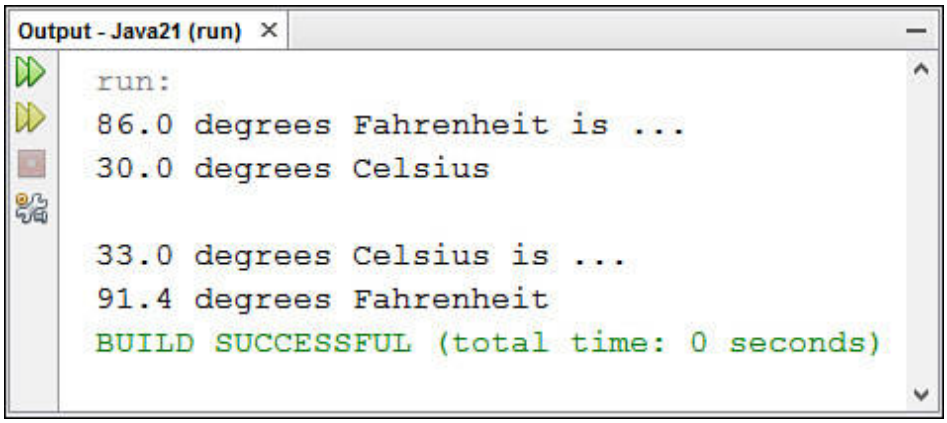


FIGURE 2.2 Converting temperatures with expressions.

In lines 5–14 of this Java application, a temperature in Fahrenheit is converted to Celsius using the arithmetic operators:

- **Line 5**—The floating-point variable `fah` is created with a value of 86.
- **Line 6**—The current value of `fah` is displayed.
- **Line 7**—The first of several comments explains what the program is doing. The Java compiler ignores these comments.
- **Line 9**—`fah` is set to its current value minus 32.
- **Line 11**—`fah` is set to its current value divided by 9.
- **Line 13**—`fah` is set to its current value multiplied by 5.
- **Line 14**—Now that `fah` has been converted to a Celsius value, `fah` is displayed again.

A similar thing happens in lines 16–25, but in the reverse direction. A temperature in Celsius is converted to Fahrenheit.

More About Assignment

Assigning a value to a variable is an expression because it produces a value. Because of this feature, you can combine assignment statements in this unusual way:

```
x = y = z = 7;
```

In this statement, all three variables `x`, `y`, and `z` end up with the value 7.

The right side of an assignment expression always is calculated before the assignment takes place. This makes it possible to use an expression statement as in the following code:

```
int x = 5;
x = x + 2;
```

In the expression `x = x + 2`, the first thing that happens is that `x + 2` is calculated. The result of this calculation, 7, is then assigned to `x`.

Using an expression to change a variable's value is a common task in programming. Several operators are used strictly in these cases.

Table 2.4 shows these assignment operators and the expressions they are functionally equivalent to.

Expression	Meaning
`x += y`	`x = x + y`
`x -= y`	`x = x – y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`

TABLE 2.4 Assignment Operators

Caution

These shorthand assignment operators are functionally equivalent to the longer assignment statements for which they substitute. If either side of your assignment statement is part of a complex expression, however, there are cases where the operators are not equivalent. For example, if `x` equals 20 and `y` equals 5, the following two statements do not produce the same value:

```
x = x / y + 5;
x /= y + 5;
```

The first statement produces an `x` value of 9 and the second an `x` value of 2. When in doubt about what an expression is doing, simplify it by using multiple assignment statements and don't use the shorthand operators.

Incrementing and Decrementing

Another common task required in programming is to add or subtract 1 from an integer variable. These expressions have special operators, which are called increment and decrement operators. *Incrementing* a variable means adding 1 to its value, and *decrementing* a variable means subtracting 1 from its value.

The increment operator is `++`, and the decrement operator is `--`. These operators are placed immediately after or before a variable name, as in the following code:

```
int x = 7;
x++;
```

In this example, the statement `x++` increments the `x` variable from 7 to 8.

These increment and decrement operators can be placed before or after a variable name. This affects the value of expressions that involve these operators.

Increment and decrement operators are called *prefix* operators if listed before a variable name and *postfix* operators if listed after a name.

In a simple expression such as `count--;`, using a prefix or postfix operator produces the same result, making the operators interchangeable. When increment and decrement operations are part of a larger expression, however, the choice between prefix and postfix operators is important.

Consider the following code:

```
int x, y, z;
x = 42;
y = x++;
z = ++x;
```

The three expressions in this code yield different results because of the difference between prefix and postfix operations.

When you use postfix operators on a variable in an expression, the variable's value is evaluated in the expression before it is incremented or decremented. So in `y = x++`, `y` receives the value of `x` before it is incremented by 1.

When using prefix operators on a variable in an expression, the variable is incremented or decremented before its value is evaluated in that expression. Therefore, in `z = ++x`, `x` is incremented by 1 before the value is assigned to `z`.

The end result of the preceding codes example is that `y` equals 42, `z` equals 44, and `x` equals 44.

If you're still having some trouble figuring this out, here's the example again with comments describing each step:

Click here to view code image

```
int x, y, z; // x, y, and z are declared
x = 42;      // x is given the value 42
y = x++;     // y is given x's value (42) before it is
incremented
             // and x is then incremented to 43
z = ++x;     // x is incremented to 44, and z is given x's value
```

Caution

Using increment and decrement operators in complex expressions can produce results you might not expect.

The concept of "assigning `x` to `y` before `x` is incremented" isn't precisely right, because Java evaluates everything on the right side of an expression before assigning its value to the left side.

Java stores some values before handling an expression to make postfix work the way it has been described in this section.

If you're not getting the results you expect from a complex expression that includes prefix and postfix operators, try breaking the expression into multiple statements to simplify it.

Comparisons

Java has several operators for making comparisons among variables, variables and literals, or other types of information in a program.

These operators are used in expressions that return Boolean values of `true` or `false`, depending on whether the comparison being made is true or not. Table 2.5 shows the comparison operators.

Operator	Meaning	Example
==	Equal to	x == 3
!=	Not equal to	x != 3
<	Less than	x < 3
>	Greater than	x > 3
<=	Less than or equal to	x <= 3
>=	Greater than or equal to	x >= 3

TABLE 2.5 Comparison Operators

The following example shows a comparison operator in use:

```
boolean isHip;
int age = 37;
isHip = age < 25;
```

The expression `age < 25` produces a result of either `true` or `false`, depending on the value of the integer `age`. Because `age` is 37 in this example (which is not less than 25), `isHip` is given the Boolean value `false`.

Logical Operators

Expressions that result in Boolean values, such as comparison operations, can be combined to form more complex expressions. This is handled through logical operators, which are used for the logical combinations `AND`, `OR`, `XOR`, and logical `NOT`.

For `AND` combinations, the `&` or `&&` logical operator is used. When two Boolean expressions are linked by these operators, the combined expression returns a `true` value only if both Boolean expressions are true.

Consider this example:

Click here to view code image

```
boolean extraLife = (score > 75000) & (playerLives < 10);
```

This expression combines two comparison expressions: `score > 75000` and `playerLives < 10`. If both expressions are true, the Boolean value `true` is assigned to the variable `extraLife`. In any other circumstance, the value `false` is assigned to the variable.

The difference between `&` and `&&` lies in how much work Java does on the combined expression. If `&` is used, the expressions on both sides of the `&` are

evaluated no matter what. If `&&` is used and the left side of the `&&` is false, the expression on the right side of the `&&` never is evaluated.

This makes `&&` more efficient because no unnecessary work is performed. In the preceding example if `score` is not greater than 75,000, there's no need to consider whether `playerLives` is less than 10.

For `OR` combinations, the `|` or `||` logical operator is used. These combined expressions return a `true` value if either Boolean expression is true.

Consider this example:

Click here to view code image

```
boolean extralife = (score > 75000) || (playerLevel == 0);
```

This expression combines two comparison expressions: `score > 75000` and `playerLevel == 0`. If either of these expressions is true, the Boolean value `true` is assigned to the variable `extraLife`. Only if both of these expressions are false is the value `false` assigned to `extraLife`.

Note the use of `||` instead of `|`. Because of this usage, if `score > 75000` is true, `extraLife` is set to `true`, and the second expression never is evaluated.

The `XOR` combination has one logical operator, `^`. This results in a `true` value only if the Boolean expressions it combines have opposite values. If both are true or both are false, the `^` operator produces a `false` value.

The `NOT` combination uses the `!` logical operator followed by a single expression. It reverses the value of a Boolean expression in the same way that a minus sign reverses the positive or negative sign on a number. For example, if `age < 25` returns a `true` value, `!(age < 25)` returns a `false` value.

The logical operators may seem illogical when you first encounter them. You get plenty of opportunities to work with them during the rest of this week, especially on Day 5, "Creating Classes and Methods."

Operator Precedence

When more than one operator is used in an expression, Java has an established precedence hierarchy to determine the order in which operators are evaluated. In many cases, this precedence determines the expression's overall value.

For example, consider the following expression:

```
y = 6 + 4 / 2;
```

The `y` variable will equal the value 5 or the value 8, depending on which arithmetic operation is handled first. If the `6 + 4` expression comes first, `y` has the value of 5. Otherwise, `y` equals 8.

In general, the order of evaluation from first to last is as follows:

1. Increment and decrement operations
2. Arithmetic operations
3. Comparisons
4. Logical operations
5. Assignment expressions

If two operations have the same precedence, the one on the left in the expression is handled before the one on the right. Table 2.6 shows the specific precedence of the various operators in Java. Operators higher up in the table are evaluated first.

Operator	Notes
`. [] ()`	A period . is used for access to methods and variables within objects and classes. Square brackets [] are used for arrays. Parentheses () are used to group expressions.
`++ -- ! ~ instanceof`	The `instanceof` operator returns `true` or `false` based on whether the object is an instance of the named class or any of that class's subclasses.
`new (type)expression`	The `new` operator is used to create new instances of classes. The parentheses in this case are for casting a value to another type.
`* / %`	Multiplication, division, modulus
`+ -`	Addition, subtraction
`<< >> >>>`	Bitwise left and right shift
`< > <= >=`	Relational comparison tests
`== !=`	Equality
`&`	AND
`^`	XOR
`|`	OR
`&&`	Logical AND
`||`	Logical OR
`? :`	Ternary operator
`= += -= *= /= %= ^=`	Various assignments
`&= |= <<= >>= >>>=`	More assignments

TABLE 2.6 Operator Precedence

Several of the operators listed in Table 2.6 are covered later this week.

Returning to the expression `y = 6 + 4 / 2`, Table 2.6 shows that division is evaluated before addition, so the value of `y` equals 8.

To change the order in which expressions are evaluated, place parentheses around the expressions that should be evaluated first. You can nest one set of parentheses inside another to make sure that expressions are evaluated in the desired order; the innermost parenthetic expression is evaluated first.

The following expression results in a value of 5:

```
y = (6 + 4) / 2
```

The value of 5 is the result because `6 + 4` is calculated first, and then the result, 10, is divided by 2.

Parentheses also can improve an expression's readability. If an expression's precedence isn't immediately clear to you, adding parentheses to impose the desired precedence can make the statement easier to understand.

String Arithmetic

As stated earlier, the + operator has a double life outside the world of mathematics. It can concatenate two or more strings.

The word "concatenate" means to link two things. For reasons unknown, it is the verb of choice in computer programming when describing the act of combining two strings, winning out over paste, glue, affix, combine, link, smush together, and conjoin.

In several examples, you have seen statements that look something like this:

Click here to view code image

```
String brand = "Jif";
System.out.println("Choosy mothers choose " + brand);
```

These two lines result in the display of the following text:

```
Choosy mothers choose Jif
```

The + operator combines strings, other objects, and variables to form a single string. In the preceding example, the literal "Choosy mothers choose" is concatenated to the value of the `String` object `brand`.

Working with the concatenation operator is made easier in Java by the fact that the operator can handle any variable type and object value as if it were a string. If any part of a concatenation operation is a `String` or a string literal, all elements of the operation are treated as if they were strings:

Click here to view code image

```
System.out.println(4 + " score and " + 7 + " years ago");
```

This produces the output text "4 score and 7 years ago", as if the integer literals 4 and 7 were strings.

There also is a += shorthand operator to append something to the end of a string. For example, consider the following expression:

```
myName += " Jr.";
```

This expression is equivalent to the following:

```
myName = myName + " Jr.";
```

In this example, += changes the value of `myName`, which might be something like “Robert Downey,” by adding “Jr.” at the end to form the string “Robert Downey Jr.”

To summarize today’s material, Table 2.7 lists the operators you have learned about. Be a doll and look them over carefully.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&&	Logical AND
||	Logical OR
!	Logical NOT
&	AND
|	OR
^	XOR
=	Assignment
++	Increment
--	Decrement
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign

TABLE 2.7 Operator Summary

Summary

Anyone who pops open a set of matryoshka dolls has to be a bit disappointed upon reaching the smallest doll in the group.

Today you reached Java's smallest nesting doll. Using statements and expressions enables you to begin building effective methods, which makes effective objects and classes possible.

Today you learned about creating variables and assigning values to them. You also used literals to represent numeric, character, and string values and worked with operators. Tomorrow, you'll put these skills to use developing classes.

Q&A

Q What happens if I assign an integer value to a variable that is too large for that variable to hold?

A Logically, you might think that the variable is converted to the next-larger type, but this isn't what happens. Instead, an *overflow* occurs—a situation in which the number wraps around from one size extreme to the other. An example of overflow would be a `byte` variable that goes from 127 (an acceptable value) to 128 (unacceptable). It would wrap around to the lowest acceptable value, which is –128, and start counting upward from there. Overflow isn't something you can readily detect in a program, so be sure to give your numeric variables plenty of living space in their chosen data type.

Small data types like `byte` were more necessary when computers had much less memory than they do today and every byte counted. Today, with plentiful memory and hard disk space measured in terabytes, it is better to use larger data types like `int` to ensure that you have enough space to store all possible values in a particular variable.

Q Why does Java have all these shorthand operators for arithmetic and assignment? It's really hard to read that way.

A Java's syntax is based on C++, which is based on C (more Russian nesting doll behavior). C is an expert language that values programming power over readability, and the shorthand operators are one of the legacies of that design priority. Using them in a program isn't required because effective substitutes are available, so you can avoid them in your own programming if you prefer.

Q The `MarsRobot` and `MarsApplication` programs on Day 1 didn't contain a `package` statement. Does that mean they're not in

a package?

A All Java programs belong to a package. When the `package` statement appears in a program, the program is part of that named package. The programs you created during this day are all in the `com.java21days` package.

A program that does not have a `package` statement is put into the default package, which does not have a name. Although programs can be created in this unnamed package, it's good practice to always specify a package with each program you create in Java. (This wasn't done in Day 1 for the sake of simplicity.)

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which of the following is a valid value for a `boolean` variable?

   A. "false"

   B. `false`

   C. 10

2. Which of these is NOT a convention for naming variables in Java?

   A. After the first word in the variable name, each successive word begins with a capital letter.

   B. The first letter of the variable name is lowercase.

   C. All letters are capitalized.

3. Which of these data types holds numbers from –32,768 to 32,767?

   A. `char`

   B. `byte`

   C. `short`

Answers

1. B. In Java, a `boolean` can be only `true` or `false`. If you put quotation marks around the value, it is treated like a `String` rather than one of the two `boolean` values.

2. C. Constant names are capitalized to make them stand out from other variables.

3. C. The `short` primitive data type has that range of values.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material.

Which of the following data types can hold the number 3,000,000,000 (3 billion)?

A. `short`, `int`, `long`, `float`

B. `int`, `long`, `float`

C. `long, float`

D. `byte`

The answer is available on the book's website at www.java21days.com. Visit the Day 2 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a program that calculates how much a $14,000 investment would be worth if it increased in value by 40% during the first year, lost $1,500 in value the second year, and increased 12% in the third year.
2. Write a program that displays two numbers and uses the `/` and `%` operators to display the result and remainder after they are divided. Use the `\t` character escape code to make a tab character separate the result and remainder in your output.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 3. Working with Objects

Java is an object-oriented programming language. When you do work in Java, you primarily use objects to get the job done. You create objects, modify them, change their variables, call their methods, and combine them with other objects. You develop classes, create objects out of those classes, and use them with other classes and objects.

Today, you work extensively with objects as you undertake these essential tasks:

- Creating objects
- Testing and modifying their class and instance variables
- Calling an object's methods
- Converting objects from one class to another

Creating New Objects

When you write a Java program, you define a set of classes. As you learned during Day 1, "Getting Started with Java," a class is a template used to create one or more objects. These objects, which also are called *instances*, are self-contained elements of a program with related features and data. For the most part, you use the class merely to create instances and then work with those instances. In this section, you learn how to create a new object from any given class.

When using strings during Day 2, "The ABCs of Programming," you learned that a string literal (a series of characters enclosed in double quotation marks) can be used to create a new instance of the class `String` with the value of that string.

The `String` class is unusual in that respect. Although it's a class, it can be assigned a value with a literal as if it was a primitive data type. This shortcut is available only for strings and classes that represent primitive data types, such as `Integer` and `Double`. To create instances for all other classes, the `new` operator is used.

Note

What about the literals for numbers and characters? Don't they create objects too? Actually, they don't. The primitive data types for numbers and characters create numbers and characters, but for efficiency they actually aren't objects. On Day 5, "Creating Classes and Methods," you learn how to use objects to represent primitive values.

Using `new`

To create a new object, you use the `new` operator with the name of the class that should be used as a template. The name of the class is followed by parentheses, as in these three examples:

Click here to view code image

```
String name = new String("Hal Jordan");

URL address = new URL("http://www.java21days.com");

MarsRobot robbie = new MarsRobot();
```

The parentheses are important and can't be omitted. They can be empty, however, in which case the most simple, basic object of that class is created. The parentheses also can contain arguments that determine the values of instance variables or other initial qualities of that object.

Here are two objects being created with arguments:

Click here to view code image

```
Random seed = new Random(606843071);

Point pt = new Point(0, 0);
```

The number and type of arguments to include inside the parentheses are defined by the class itself using a special method called a *constructor* (which is introduced later today). If you try to create a new instance of a class with the wrong number or wrong type of arguments, or if you give it no arguments and it needs them, an error occurs when the program is compiled.

Today's first project is a demonstration of creating different types of objects with different numbers and types of arguments. The `StringTokenizer` class in the `java.util` package divides a string into a series of shorter strings called *tokens*.

You divide a string into tokens by applying a character or characters as a delimiter. For example, the text "02/20/67" could be divided into three tokens —"02", "20", and "67"—using the slash character / as a delimiter.

Today's first project is a Java application that uses string tokens to analyze stock price data. In NetBeans, create a new empty Java file for the class `TokenTester` in the `com.java21days` package, and enter the code in Listing 3.1 as its source code. This program creates `StringTokenizer` objects by using `new` in two different ways and then displays each token the objects contain.

LISTING 3.1 The Full Text of `TokenTester.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.util.StringTokenizer;
 4:
 5: class TokenTester {
 6:
 7:     public static void main(String[] arguments) {
 8:         StringTokenizer st1, st2;
 9:
10:         String quote1 = "GOOG 530.80 -9.98";
11:         st1 = new StringTokenizer(quote1);
12:         System.out.println("Token 1: " + st1.nextToken());
13:         System.out.println("Token 2: " + st1.nextToken());
14:         System.out.println("Token 3: " + st1.nextToken());
15:
16:         String quote2 = "RHT@75.00@0.22";
17:         st2 = new StringTokenizer(quote2, "@");
18:         System.out.println("\nToken 1: " + st2.nextToken());
19:         System.out.println("Token 2: " + st2.nextToken());
20:         System.out.println("Token 3: " + st2.nextToken());
21:     }
22: }
```

Save this file by choosing File, Save or clicking Save All on the NetBeans toolbar. Run the application by choosing Run, Run File to see the output displayed in Figure 3.1.

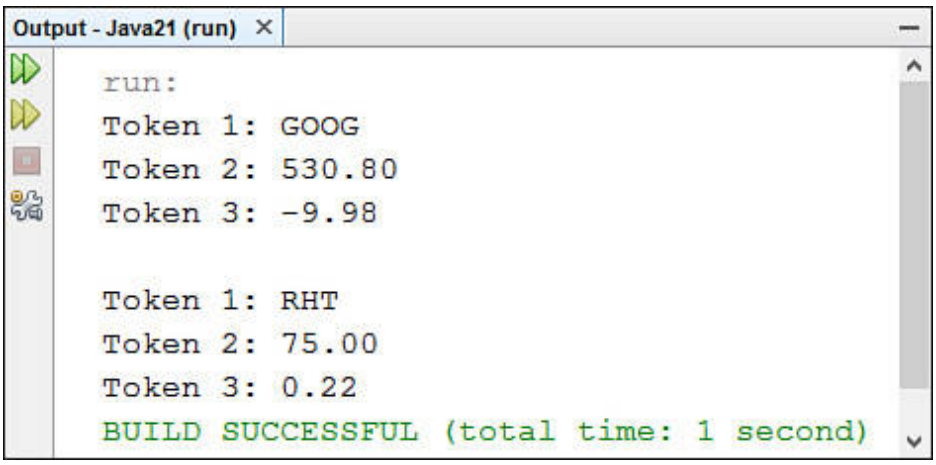


FIGURE 3.1 Displaying a `StringTokenizer` object's tokens.

Two different `StringTokenizer` objects are created using different arguments to the constructor.

The first object is created using `new StringTokenizer()` with one argument, a `String` object named `quote1` (line 11). This creates a `StringTokenizer` object that uses the default delimiters, which are blank spaces, tabs, newlines, carriage returns, or formfeed characters.

If any of these characters is contained in the string, it is used to divide the string. Because the `quote1` string contains spaces, these are used as delimiters dividing each token. Lines 12–14 display the values of all three tokens: "GOOG", "530.80", and "–9.98".

The second `StringTokenizer` object in this example has two arguments when it is constructed in line 16—a `String` object named `quote2` and an at-sign character @. This second argument indicates that the @ character should be used as the delimiter between tokens. The `StringTokenizer` object created in line 17 contains three tokens: "RHT", "75.00", and "0.22".

How Objects Are Constructed

Several things happen when you use the `new` operator. The new instance of the given class is created, memory is allocated for it, and a special method defined in the given class is called. This method is called a constructor.

A *constructor* is a way to create a new instance of a class. A constructor initializes the new object and its variables, creates any other objects that the object needs, and performs any additional operations the object requires to initialize itself.

A class can have several different constructors, each with a different number or type of argument. When you use `new`, you can specify different arguments in the argument list, and the correct constructor for those arguments is called.

In the TokenTester project, multiple constructor definitions enabled the `StringTokenizer` class to accomplish different things with different uses of the `new` operator. When you create your own classes, you can define as many constructors as you need to implement the behavior of the class.

No two constructors in a class can have the same number and type of arguments, because this is the only way constructors are differentiated from each other.

If a class defines no constructors, a constructor with no arguments is called by default when an object of the class is created. The only thing this constructor does is call the same constructor in its superclass.

Caution

The default constructor only exists in a class that has not defined any constructors. Once you define at least one constructor in a class, you can't count on there being a default constructor with no arguments.

A Note on Memory Management

If you are familiar with other object-oriented programming languages, you might wonder whether the `new` operator has an opposite that destroys an object when it is no longer needed.

Memory management in Java is dynamic and automatic. When you create a new object, Java automatically allocates the proper amount of memory for that object. You don't have to allocate any memory for objects explicitly. The Java Virtual Machine (JVM) does it for you.

Because Java memory management is automatic, you don't need to deallocate the memory an object uses when you're finished using it. Under most circumstances, when you are finished with an object you have created, Java can determine that the object no longer has any live references to it. (In other words, the object isn't assigned to any variables still in use or stored in any arrays.)

As a program runs, the JVM periodically looks for unused objects and reclaims the memory that those objects are using. This process is called

dynamic garbage collection and occurs without any programming on your part. You don't have to explicitly free the memory taken up by an object; you just have to make sure that you're not still holding onto an object you want to get rid of.

This feature is one of the most touted advantages of the language over its predecessor C++.

Using Class and Instance Variables

At this point, you can create your own object with class and instance variables, but how do you work with those variables? They're used in largely the same manner as the local variables you learned about yesterday. You can put them in expressions, assign values to them in statements, and so on. You just refer to them slightly differently.

Getting Values

To get to the value of an instance variable, you use *dot notation*, a form of addressing in which an instance or class variable name has two parts:

- A reference to an object or class on the left side of a dot operator.
- A variable on the right side

Dot notation is how you refer to an object's instance variables and methods.

For example, if you have an object named `customer` with a variable called `orderTotal`, here's how that variable could be referred to in a statement:

Click here to view code image

```
float total = customer.orderTotal;
```

This statement assigns the value of the `customer` object's `orderTotal` instance variable to a floating-point variable named `total`.

Accessing variables in dot notation is an expression (meaning that it returns a value). Both sides of the dot also are expressions. This means that you can chain instance variable access.

Extending the preceding example, suppose the `customer` object is an instance variable of the `store` class. Dot notation can be used twice, as in this statement:

Click here to view code image

```
float total = store.customer.orderTotal;
```

Dot expressions are evaluated from left to right, so you start with `store`'s instance variable `customer`, which itself has an instance variable `orderTotal`. The value of this variable is assigned to the `total` variable.

One thing to note when chaining objects together in this manner is that the statement will fail with an error if any object in the chain does not have a value yet.

Setting Values

Assigning a value to an instance variable with dot notation employs the = operator just like variables holding primitive types:

```
customer.layaway = true;
```

This example sets the value of a `boolean` instance variable named `layaway` to `true`.

The PointSetter application in Listing 3.2 tests and modifies the instance variables in a `Point` object. `Point`, a class in the `java.awt` package, represents points in a coordinate system with (x, y) values.

Create a new empty Java file in NetBeans with the class name `PointSetter` and the package name `com.java21days`; then type the source code shown in Listing 3.2 and save the file.

LISTING 3.2 The Full Text of `PointSetter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class PointSetter {
 6:
 7:     public static void main(String[] arguments) {
 8:         Point location = new Point(4, 13);
 9:
10:         System.out.println("Starting location:");
11:         System.out.println("X equals " + location.x);
12:         System.out.println("Y equals " + location.y);
13:
14:         System.out.println("\nMoving to (7, 6)");
15:         location.x = 7;
16:         location.y = 6;
```

```
17:
18:         System.out.println("\nEnding location:");
19:         System.out.println("X equals " + location.x);
20:         System.out.println("Y equals " + location.y);
21:     }
22: }
```

When you run this application, the output should match Figure 3.2.

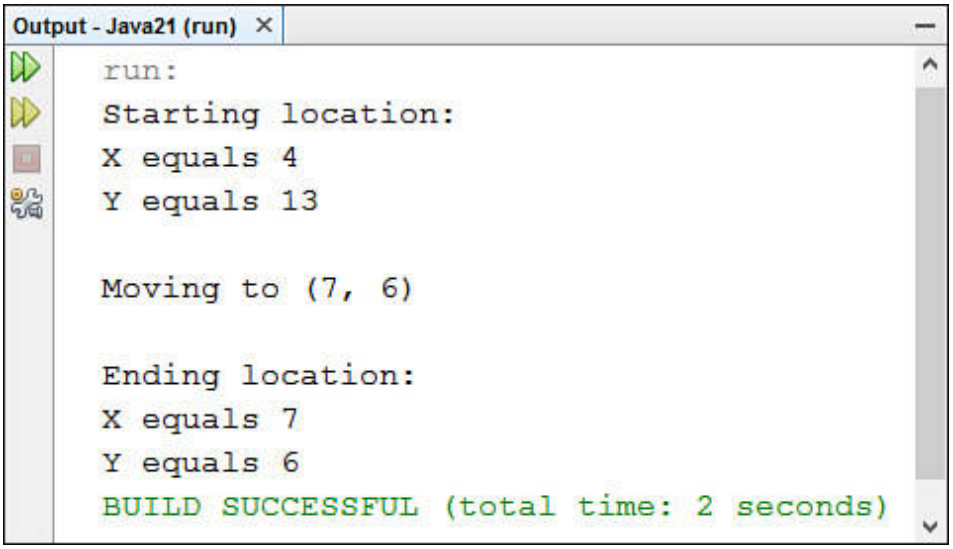


FIGURE 3.2 Setting and displaying an object's instance variables.

In this application, you create an instance of `Point` where `x` equals 4 and `y` equals 13 (line 8). These individual values are retrieved using dot notation.

The value of `x` is changed to 7 and `y` to 6 (lines 15–16). The values are displayed again to show how they have changed.

Class Variables

Class variables, as you have learned, are variables defined and stored in the class itself. Their values apply to the class and all its instances.

With instance variables, each new instance of the class gets a new copy of the instance variables that the class defines. Each instance then can change the values of those instance variables without affecting any other instances. With class variables, only one copy of that variable exists when the class is loaded. Changing the value of that variable changes it for all instances of that class.

You define class variables by including the `static` keyword before the variable itself. For example, consider the following partial class definition:

Click here to view code image

```
class FamilyMember {
    static String surname = "Mendoza";
    String name;
    int age;
}
```

Each instance of the class `FamilyMember` has its own values for `name` and `age`, but the class variable `surname` has only one value for all family members: “Mendoza.” If the value of `surname` is changed, all instances of `FamilyMember` are affected.

Note

Calling these static variables refers to one of the meanings of the word “static”: fixed in one place. If a class has a `static` variable, every object of that class has the same value for that variable.

To access class variables, you use the same dot notation as with instance variables. To retrieve or change the value of the class variable, you can use either the instance or the name of the class on the left side of the dot operator. Both lines of output in this example display the same value:

Click here to view code image

```
FamilyMember dad = new FamilyMember();
System.out.println("Family's surname is: " + dad.surname);
System.out.println("Family's surname is: " +
FamilyMember.surname);
```

Because you can use an object to change the value of a class variable, it’s easy to become confused about class variables and where their values are coming from. Remember that the value of a class variable affects all objects of that particular class. If the `surname` instance variable of one FamilyMember object was set to “Paciorek”, all objects of that class would have that new surname.

To reduce confusion when using class variables, it’s a good idea to use the name of the class when you refer to a class variable—not an object of that class. This makes the use of a class variable more clear and helps strange results become easier to debug.

Calling Methods

Methods of an object are called to make it do something.

Calling a method in an object also makes use of dot notation. The object whose method is being called is on the left side of the dot, and the name of the method and its arguments are on the right side:

Click here to view code image

```
customer.addToCart(itemNumber, price, quantity);
```

All method calls must have parentheses after them, even when the method takes no arguments, as in this example:

```
customer.cancelOrder();
```

In Listing 3.3, the StringChecker application shows an example of calling some methods defined in the `String` class. Strings include methods for string tests and modification. Create this program in NetBeans as an empty Java file with the class name `StringChecker` and package name `com.java21days`.

LISTING 3.3 The Full Text of `StringChecker.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class StringChecker {
 4:
 5:     public static void main(String[] arguments) {
 6:         String str = "A Lannister always pays his debts";
 7:         System.out.println("The string is: " + str);
 8:         System.out.println("Length of this string: "
 9:             + str.length());
10:         System.out.println("The character at position 6: "
11:             + str.charAt(6));
12:         System.out.println("The substring from 12 to 18: "
13:             + str.substring(12, 18));
14:         System.out.println("The index of the first 't': "
15:             + str.indexOf('t'));
16:         System.out.println("The index of the beginning of the
"
17:             + "substring \"debts\": " +
str.indexOf("debts"));
18:         System.out.println("The string in uppercase: "
19:             + str.toUpperCase());
20:     }
21: }
```

Running the program produces the output shown in Figure 3.3.

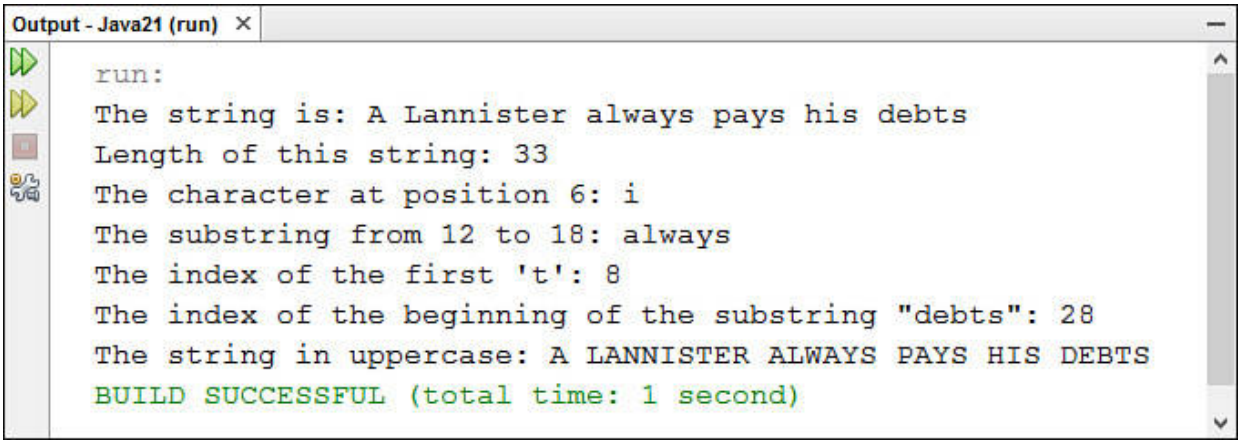


FIGURE 3.3 Calling `String` methods to learn more about that string.

In line 6, you create a new instance of `String` by using the string literal "A Lannister always pays his debts". The remainder of the program simply calls different string methods to do different operations on that string:

- Line 7 prints the value of the string.
- Line 9 calls the `length()` method in the new `String` object to find out how many characters it contains.
- Line 11 calls the `charAt()` method, which returns the character at the given position in the string. Note that string positions start at position 0 rather than 1, so the character at position 6 is 'i'.
- Line 13 calls the `substring()` method, which takes two integers indicating a range and returns the substring with those starting and ending points. The `substring()` method also can be called with only one argument, which returns the substring from that position to the end of the string.
- Line 15 calls the `indexOf()` method, which returns the position of the first instance of the given character. Character literals are surrounded by single quotation marks, so the argument is 't' (not "t").
- Line 17 shows a different use of the `indexOf()` method, which takes a string argument and returns the index of the beginning of that string. String literals always are surrounded by double quotation marks.
- Line 19 uses the `toUpperCase()` method to return a copy of the string in all uppercase.

Note

If you compare the output of the `StringChecker` application to the characters in the string, you might be wondering how 'i' could be at position 6 when it is the seventh character in the string. All of the methods look like they're off by one (except for `length()`). The reason is that the methods are zero-based, which means they begin counting with 0 instead of 1. So 'A' is at position 0, a space at position 1, 'L' at position 2 and so on. This kind of numbering is something you encounter often in Java.

Formatting Strings

Numbers such as money often need to be displayed in a precise manner. There are only two places after the decimal for the number of cents, a dollar sign ($) preceding the value, and commas separating groups of three numbers—as in $22,453.70 (the amount the U.S. National Debt goes up in one second).

This kind of formatting when displaying strings can be accomplished with the `System.out.format()` method.

The method takes two arguments: the output format template and the string to display. Here's an example that adds a dollar sign and commas to the display of an integer:

Click here to view code image

```
int accountBalance = 5005;
System.out.format("Balance: $%,d%n", accountBalance);
```

This code produces the output `Balance: $5,005`.

The formatting string begins with a percent sign % followed by one or more flags. The `%,d` code displays a decimal with commas dividing each group of three digits. The `%n` code displays a newline character.

The next example displays the value of pi to 11 decimal places:

Click here to view code image

```
double pi = Math.PI;
System.out.format("%.11f%n", pi);
```

The output is `3.14159265359`.

Tip

Oracle's Java site includes a beginner's tutorial for `printf`-style output that describes some of the most useful formatting codes:

http://docs.oracle.com/javase/tutorial/java/data/numberformat.html

Nesting Method Calls

A method can return a reference to an object, a primitive data type, or no value at all. In the StringChecker application, all the methods called on the `String` object `str` returned values that are displayed. The `charAt()` method returned a character at a specified position in the string.

The value returned by a method also can be stored in a variable:

Click here to view code image

```
String label = "From";
String upper = label.toUpperCase();
```

In this example, the `String` object `upper` contains the value returned by calling `label.toUpperCase()`, which is the text "FROM".

If the method returns an object, you can call the methods of that object in the same statement. This makes it possible for you to nest methods as you would variables.

Earlier today, you saw an example of a method called with no arguments:

```
customer.cancelOrder();
```

If the `cancelOrder()` method returns an object, you can call methods of that object in the same statement:

Click here to view code image

```
customer.cancelOrder().fileComplaint();
```

This statement calls the `fileComplaint()` method, which is defined in the object returned by the `cancelOrder()` method of the `customer` object.

You can combine nested method calls and instance variable references as well. In the next example, the `putOnLayaway()` method is defined in the object stored by the `orderTotal` instance variable, which itself is part of the `customer` object:

Click here to view code image

```
customer.orderTotal.putOnLayaway(itemNumber, price, quantity);
```

This manner of nesting variables and methods is demonstrated in a method you've used frequently in the first three days of this book: `System.out.println()`.

That method displays strings and other data to the computer's standard output device.

The `System` class, part of the `java.lang` package, describes behavior specific to the computer system on which Java is running. `System.out` is a class variable that contains an instance of the class `PrintStream` representing the system's standard output, which normally is the monitor but can be a printer or file. `PrintStream` objects have a `println()` method that sends a string to that output stream. The `PrintStream` class is in the `java.io` package.

Class Methods

Class methods, also called static methods, apply to the class as a whole and not to its instances just like class variables. Class methods commonly are used for general utility methods that might not operate directly on an object of that class but do fit with that class conceptually.

For example, the `String` class contains a class method called `valueOf()`, which can take one of many types of arguments (integers, Booleans, objects, and so on). The `valueOf()` method then returns a new instance of `String` containing the argument's string value. This method doesn't operate directly on an existing instance of `String`, but getting a string from another object or data type is behavior that makes sense to define in the `String` class.

Class methods also can be useful for gathering general methods in one place. For example, the `Math` class, defined in the `java.lang` package, contains a large set of mathematical operations as class methods. No objects can be created from the `Math` class, but you still can use its methods with numeric or Boolean arguments.

For example, the class method `Math.max()` takes two arguments and returns the larger of the two. You don't need to create a new instance of `Math`; it can be called anywhere you need it, as in the following:

Click here to view code image

```
int firstPrice = 225;
int secondPrice = 217;
int higherPrice = Math.max(firstPrice, secondPrice);
```

Dot notation is used to call a class method. As with class variables, you can use either an instance of the class or the class itself on the left side of the dot. For the same reasons noted earlier about class variables, using the name of the class makes your code easier to read.

The last two lines in this example both produce strings equal to “550”:

```
String s, s2;
s = "potrzebie";
s2 = s.valueOf(550);
s2 = String.valueOf(550);
```

References to Objects

As you work with objects, it’s important to understand references. A *reference* is an address that indicates where an object’s variables and methods are stored.

You aren’t actually using objects when you assign an object to a variable or pass an object to a method as an argument. You aren’t even using copies of the objects. Instead, you’re using references to those objects.

To better illustrate what this means, the RefTester application in Listing 3.4 shows how references work. Create an empty Java file in NetBeans for the class `RefTester` in the package `com.java21days`, and enter Listing 3.4 as the application’s source code.

LISTING 3.4 The Full Text of `RefTester.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class RefTester {
 6:     public static void main(String[] arguments) {
 7:         Point pt1, pt2;
 8:         pt1 = new Point(100, 100);
 9:         pt2 = pt1;
10:
11:         pt1.x = 200;
12:         pt1.y = 200;
```

```
13:          System.out.println("Point1: " + pt1.x + ", " +
pt1.y);
14:          System.out.println("Point2: " + pt2.x + ", " +
pt2.y);
15:      }
16: }
```

Save and run the application. The output is shown in Figure 3.4.

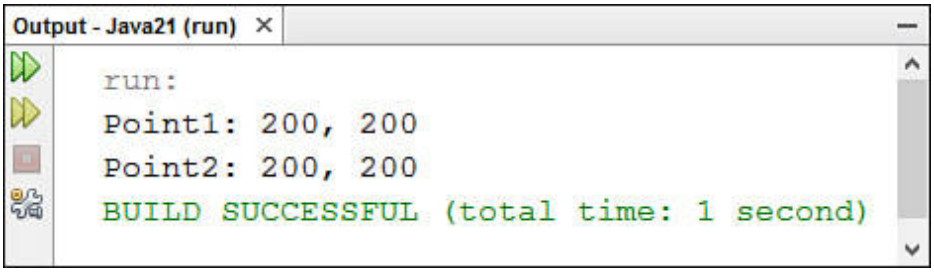


FIGURE 3.4 Putting references to a test.

The following takes place in the first part of this program:

- **Line 7**—Two `Point` variables are created.
- **Line 8**—A new `Point` object is assigned to `pt1`.
- **Line 9**—The variable `pt1` is assigned to `pt2`.

Lines 11–14 are the tricky part. The `x` and `y` variables of `pt1` both are set to 200 and all variables of `pt1` and `pt2` are displayed onscreen.

You might expect `pt1` and `pt2` to have different values, but Figure 3.4 shows this not to be the case. The `x` and `y` variables of `pt2` also have changed even though nothing in the program explicitly changes them. This happens because line 7 creates a reference from `pt2` to `pt1` instead of creating `pt2` as a new object copied from `pt1`.

The variable `pt2` is a reference to the same object as `pt1`, as shown in Figure 3.5. Either variable can be used to refer to the object or to change its variables.

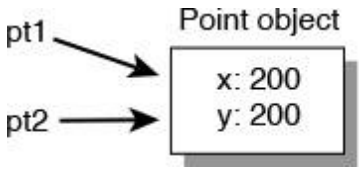


FIGURE 3.5 References to objects.

If you wanted `pt1` and `pt2` to refer to separate objects, you could use separate `new Point()` statements on lines 6–7 to create separate objects,

as shown here:

```
pt1 = new Point(100, 100);
pt2 = new Point(100, 100);
```

References in Java become particularly important when arguments are passed to methods. You learn more about this later today.

Note

Java has no explicit pointers or pointer arithmetic, unlike C and C++. By using references and Java arrays, you can duplicate most pointer capabilities without many of their drawbacks.

Casting Objects and Primitive Types

One thing you discover quickly about Java is how finicky it is about the information it will handle. Like Goldilocks, the child who is oddly hard to please about porridge for a person who breaks into homes, Java methods and constructors require things to take a specific form and won't accept alternatives.

When you send arguments to methods or use variables in expressions, you must use variables of the correct data types. If a method requires an `int`, the Java compiler responds with an error if you try to send a `float` value to the method. Likewise, if you set up one variable with the value of another, they must be of compatible types. The two variables must be the same type or the variable receiving the value must be big enough to hold the value.

Note

There is one area where Java's compiler is decidedly flexible: the `String` object. String handling in `println()` methods, assignment statements, and method arguments is simplified by the + concatenation operator. If any variable in a group of concatenated variables is a string, Java treats the whole thing as a `String`. This makes the following possible:

Click here to view code image

```
float gpa = 2.25F;
System.out.println("Honest, mom, my GPA is a " +
(gpa + 1.5));
```

Using the concatenation operator, a single string can hold the text representation of multiple objects and primitive types in Java.

Sometimes you have a value in your Java class that isn't the right type for what you need. It might be the wrong class or the wrong data type, such as a `float` when you need an `int`.

In these situations, you can use a process called *casting* to convert a value from one type to another.

Although casting is reasonably simple, the process is complicated by the fact that Java has both primitive types (such as `int`, `float`, and `boolean`) and object types (`String`, `Point`, and the like). This section discusses three forms of casts and conversions:

- Casting between primitive types, such as `int` to `float` or `float` to `double`
- Casting from an object of a class to an object of another class, such as from `Object` to `String`
- Casting primitive types to objects and then extracting primitive values from those objects

When discussing casting, it can be easier to think in terms of sources and destinations. The source is the variable being cast into another type. The destination is the result.

Casting Primitive Types

Casting between primitive types enables you to convert the value of one type to another. This most commonly occurs with the numeric types, but one primitive type never can be used in a cast. Boolean values must be either `true` or `false` and cannot be used in a casting operation.

In many casts between primitive types, the destination can hold larger values than the source, so the value is converted easily. An example would be casting a `byte` into an `int`. Because a `byte` holds values from –128 to 127 and an `int` holds from around –2,100,000 to 2,100,000, there's more than enough room to cast a `byte` into an `int`.

Often you can automatically use a `byte` or `char` as an `int`; an `int` as a `long`; an `int` as a `float`; or anything as a `double`. In most cases, because the larger type provides more precision than the smaller, no loss of information occurs as a result. The exception is casting integers to floating-point values. Casting a `long` to a `float` or a `long` to a `double` can cause some loss of precision.

Note

A character can be used as an `int` because each character has a corresponding numeric code that represents its position in the character set. If the variable `key` has the value 65, the cast `(char) key` produces the character value `'A'`. The numeric code associated with a capital A is 65 in the ASCII character set, which Java adopted as part of its character support.

You must use an explicit cast to convert a value in a large type to a smaller type. Explicit casts take the following form:

```
(typename) value
```

Here *`typename`* is the name of the primitive data type to which you're converting, such as `short`, `int`, or `float`. *`value`* is an expression that results in the value of the source type. For example, in the following statement, the value of `x` is divided by the value of `y` and the result is cast into an `int`:

```
int result = (int)(x / y);
```

Note that because the precedence of casting is higher than that of arithmetic, you have to use parentheses here. Otherwise, first the value of `x` would be cast into an `int`, and then it would be divided by `y`, which could produce a different result.

Casting Objects

Objects of classes also can be cast into objects of other classes when the source and destination classes are related by inheritance and one class is a subclass of the other.

Some objects might not need to be cast explicitly. In particular, because a subclass contains all the information as its superclass, you can use an object of a subclass anywhere a superclass is expected.

For example, consider a method that takes two arguments, one of type `Object` and another of type `Component` in the `java.awt` package (which has classes for a graphical user interface).

You can pass an instance of any class for the `Object` argument because all Java classes are subclasses of `Object`.

For the `Component` argument, you can pass in its subclasses, such as `Button`, `Container`, and `Label` (all in `java.awt`).

This is true anywhere in a program, not just inside method calls. If you had a variable defined as class `Component`, you could assign objects of that class or any of its subclasses to that variable without casting.

This also is true in the reverse, so you can use a superclass when a subclass is expected. There is a catch, however: Because subclasses contain more behavior than their superclasses, a loss of precision occurs in the casting. Those superclass objects might not have all the behavior needed to act in place of a subclass object.

Consider this example: If you have an operation that calls methods in objects of the class `Integer`, using an object of its superclass `Number` won't include many methods specified in `Integer`. Errors occur if you try to call methods that the destination object doesn't have.

To use superclass objects where subclass objects are expected, you must cast them explicitly. You won't lose any information in the cast, but you gain all the methods and variables that the subclass defines.

To cast an object to another class, you use the same operation as for primitive types, which takes this form:

```
(classname) object
```

In this template, *classname* is the name of the destination class, and *object* is a reference to the source object. Casting creates a reference to the old object of the type *classname*; the old object continues to exist as it did before.

The following example casts an instance of the class `VicePresident` to an instance of the class `Employee`. `VicePresident` is a subclass of `Employee` with more information:

Click here to view code image

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep; // no cast needed for upward use
veep = (VicePresident) emp; // must cast explicitly
```

When you begin working with graphical user interfaces during Week 2, "The Java Class Library," you will see that casting one object is necessary whenever you use Java2D graphics operations. You must cast a `Graphics` object to a `Graphics2D` object before you can draw onscreen. The following example uses a `Graphics` object called `screen` to create a new `Graphics2D` object called `screen2D`:

Click here to view code image

```
Graphics2D screen2D = (Graphics2D) screen;
```

`Graphics2D` is a subclass of `Graphics` and both belong to the `java.awt` package. You'll explore this subject fully during Day 13, "Creating Java2D Graphics."

In addition to casting objects to classes, you can cast objects to interfaces, but only if an object's class or one of its superclasses actually implements the interface. Casting an object to an interface means that you can call one of that interface's methods even if that object's class does not actually implement that interface.

Converting Primitive Types to Objects and Vice Versa

One thing you can't do is cast from an object to a primitive data type, or vice versa.

Primitive types and objects are different things in Java, and you can't automatically cast between the two.

As an alternative, the `java.lang` package includes classes that correspond to each primitive data type: `Float`, `Boolean`, `Byte`, and so on. Most of these classes have the same names as the data types, except that the class names begin with a capital letter (`Short` instead of `short`, `Double` instead of `double`, and the like). Also, two classes have names that differ from the corresponding data type: `Character` is used for `char` variables and `Integer` is used for `int` variables.

Using the classes that correspond to each primitive type, you can create an object that holds the same value. The following statement creates an instance of the `Integer` class with the integer value 7801:

Click here to view code image

```
Integer dataCount = new Integer(7801);
```

After you have created an object in this manner, you can use it as you would any object (although you cannot change its value). When you want to use that value again as a primitive value, there are methods for that as well. For example, if you wanted to get an `int` value from a `dataCount` object, the following statement shows how:

Click here to view code image

```
int newCount = dataCount.intValue(); // returns 7801
```

A common translation you need in programs is converting a `String` to a numeric type, such as an integer. When you need an `int` as the result, this can be done by using the `parseInt()` class method of the `Integer` class. The `String` to convert is the only argument sent to the method, as in the following example:

Click here to view code image

```
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

The following classes can be used to work with objects instead of primitive data types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, and `Void`. These classes are commonly called *object wrappers* because they provide an object representation that contains a primitive value.

Caution

If you try to use the preceding example in a program, your program won't compile. The `parseInt()` method is designed to fail with a NumberFormatException error if the argument to the method is not a valid numeric value. To deal with errors of this kind, you must use special error-handling statements, which are introduced during Day 7, "Exceptions and Threads."

Working with primitive types and objects that represent the same values is made easier through autoboxing and unboxing, an automatic conversion process.

Autoboxing automatically converts a primitive type to an object. *Unboxing* converts in the other direction.

If you write a statement that uses an object where a primitive type is expected, or vice versa, the value is converted so that the statement executes successfully.

Here's an example of autoboxing and unboxing:

Click here to view code image

```
Float f1 = 12.5F;
Float f2 = 27.2F;
System.out.println("Lower number: " + Math.min(f1, f2));
```

The `Math.min()` method takes two `float` values as arguments, but the preceding example sends the method two `Float` objects as arguments instead.

The compiler does not report an error over this discrepancy. Instead, the `Float` objects automatically are unboxed into `float` values before being sent to the `min()` method.

Caution

Unboxing an object works only if the object has a value. If no constructor has been called to set up the object, compilation fails with an error.

Comparing Object Values and Classes

In addition to casting, you often will perform three other common tasks that involve objects:

- Comparing objects
- Finding out the class of any given object
- Testing whether an object is an instance of a given class

Comparing Objects

Yesterday, you learned about operators for comparing values—equal, not equal, less than, and so on. Most of these operators work only on primitive types, not on objects. If you try to use other values as operands, the Java compiler produces errors.

The exceptions to this rule are the == operator for equality and the != operator for inequality. When applied to objects, these operators don't do what you might first expect. Instead of checking whether one object has the same value as the other, they determine whether both sides of the operator refer to the same object.

To compare objects of a class and have meaningful results, you must implement special methods in your class and call those methods.

A good example of this is the `String` class. It is possible to have two different `String` objects that represent the same text. If you were to employ the == operator to compare these objects, however, they would be considered unequal. Although their contents match, they are not the same object.

To see whether two `String` objects have matching values, an `equals()` method of the class is used. The method tests each character in the string and returns `true` if the two strings have the same value. The EqualsTester application shown in Listing 3.5 illustrates this. Create the application with NetBeans in the `com.java21days` package and save the file, either by choosing File, Save or clicking the Save All toolbar button.

LISTING 3.5 The Full Text of `EqualsTester.java`

Click here to view code image

```
1: package com.java21days;
2:
```

```
 3: class EqualsTester {
 4:     public static void main(String[] arguments) {
 5:         String str1, str2;
 6:         str1 = "Boy, that escalated quickly.";
 7:         str2 = str1;
 8:
 9:         System.out.println("String1: " + str1);
10:         System.out.println("String2: " + str2);
11:         System.out.println("Same object? " + (str1 == str2));
12:
13:         str2 = new String(str1);
14:
15:         System.out.println("String1: " + str1);
16:         System.out.println("String2: " + str2);
17:         System.out.println("Same object? " + (str1 == str2));
18:         System.out.println("Same value? " +
str1.equals(str2));
19:     }
20: }
```

The program's output appears in Figure 3.6.

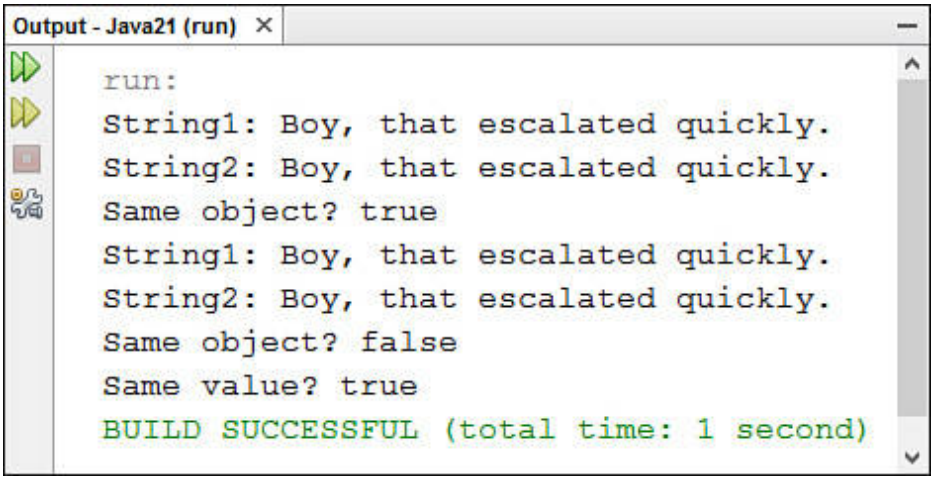


FIGURE 3.6 Calling `String` methods to learn more about that string.

The first part of this program declares two variables (`str1` and `str2`), assigns the literal "Boy, that escalated quickly." to `str1`, and then assigns that value to `str2` (lines 5–7). As you learned earlier, `str1` and `str2` now point to the same object, and the equality test at line 11 proves that.

In the second part of this program, you create a new `String` object with the same value as `str1` and assign `str2` to that new `String` object.

Now you have two different string objects in `str1` and `str2`, both with the same value. Testing them to see whether they're the same object by using the == operator returns the expected answer: `false` (line 17). They are not the

same object in memory. Testing them using the `equals()` method in line 18 also returns the expected answer of `true`, which shows they have the same value.

Note

Why can't you just use another literal when you change `str2`, instead of using `new`? String literals are optimized in Java. If you create a string using a literal and then use another literal with the same characters, Java gives you back the first `String` object. Both strings are the same object; you have to go out of your way to create two separate objects.

Determining the Class of an Object

Want to find out the name of an object's class? Here's how for an object assigned to the variable `key`:

Click here to view code image

```
String name = key.getClass().getName();
```

The `getClass()` method is defined in the `Object` class, so it can be called in all Java objects. The method returns a `Class` object that represents the object's class. That object's `getName()` method returns a string holding the name of the class.

Another useful test is the `instanceof` operator, which has two operands: a reference to an object on the left, and a class name on the right. The expression produces a Boolean value: `true` if the object is an instance of the named class or any of that class's subclasses, or `false` otherwise, as in these examples:

Click here to view code image

```
boolean check1 = "Texas" instanceof String; // true

Object obiwan = new Object();
boolean check2 = obiwan instanceof String; // false
```

The `instanceof` operator also can be used for interfaces. If an object implements an interface, the `instanceof` operator returns `true` when this is tested.

Unlike other operators in Java, `instanceof` is not a form of punctuation like * for multiplication or + for addition. Instead, the `instanceof` keyword is the operator.

Summary

Now that you have spent three days exploring how object-oriented programming is implemented in Java, you're in a better position to decide how useful it can be in your programming.

If you are a "glass half empty" kind of person, object-oriented programming (OOP) is a level of abstraction that gets in the way of using a programming language. You learn more about why OOP is thoroughly ingrained in Java in the coming days and may change your mind.

If you are a "glass half full" kind of person, object-oriented programming is beneficial because of its benefits: improved reliability, reusability, and maintenance.

Today you learned how to deal with objects: creating them, reading their values and changing them, and calling their methods. You also learned how to cast objects from one class to another, cast to and from primitive data types and classes, and take advantage of automatic conversions through autoboxing and unboxing.

Q&A

Q I'm confused about the differences between objects and the primitive data types, such as `int` and `boolean`.

A The primitive types (`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`) are not objects, although in many ways they can be handled like objects. They can be assigned to variables and passed in and out of methods.

Objects are instances of classes and as such are much more complex data types than simple numbers and characters. They often contain numbers and characters as instance or class variables.

Q The `length()` and `charAt()` methods in the StringChecker application (Listing 3.3) don't appear to make sense. If `length()` says that a string is 33 characters long, shouldn't the characters be

numbered from 1 to 33 when `charAt()` is used to display characters in the string?

A The two methods look at strings differently. The `length()` method counts the characters in the string, with the first character counting as 1, the second as 2, and so on. The `charAt()` method considers the first character in the string to be located at position number 0. This is the same numbering system used with array elements in Java. Consider the string "Charlie Brown". It has 13 characters ranging from position 0 (the letter C) to position 12 (the letter n).

Q If Java lacks pointers, how can I do something like linked lists, where there's a pointer from one node to another so that they can be traversed?

A It's incorrect to say that Java has no pointers; it just has no *explicit* pointers. Object references are effectively pointers. To create something like a linked list, you could create a class called `Node`, which would have an instance variable also of type `Node`. To link node objects, assign a node object to the instance variable of the object immediately before it in the list. Because object references are pointers, linked lists set up this way behave as you would expect them to. (You'll work with the Java class library's version of linked lists on Day 8, "Data Structures.")

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What operator do you use to call an object's constructor and create a new object?

A. +

B. `new`

C. `instanceof`

2. What kind of methods apply to all objects of a class rather than an individual object?

A. Universal methods

B. Instance methods

C. Class methods

3. If you have a program with objects named `obj1` and `obj2`, what happens when you use the statement `obj2 = obj1`?

A. The instance variables in `obj2` are given the same values as `obj1`.

B. `obj2` and `obj1` are considered to be the same object.

C. Neither A nor B.

Answers

1. B. The `new` operator is followed by a call to the object's constructor.
2. C. Class methods can be called without creating an object of that class.
3. B. The = operator does not copy values from one object to another. Instead, it makes both variables refer to the same object.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class AyeAye {
    int i = 40;
    int j;

    public AyeAye() {
        setValue(i++);
    }

    void setValue(int inputValue) {
        int i = 20;
        j = i + 1;
        System.out.println("j = " + j);
    }
}
```

What is the value of the `j` variable at the time it is displayed inside the `setValue()` method?

A. 42

B. 40

C. 21

D. 20

The answer is available on the book's website at www.java21days.com. Visit the Day 3 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a program that turns a birthday in MM/DD/YYYY format (such as 04/29/2016) into three individual strings.
2. Create a class with instance variables for `height`, `weight`, and `depth`, making each an integer. Create a Java application that uses your new class, sets each of these values in an object, and displays the values.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 4. Lists, Logic, and Loops

Today, you learn about three of the most boring features in the Java language:

- How to organize groups of the same class or data type into lists called arrays
- How to make a program decide whether to do something based on logic
- How to make part of a Java program repeat itself by using loops

If these features don't sound boring to you, that's good. Most of the significant work that you will accomplish with your Java software will use all three.

These topics are boring for computers. They enable software to do one of the things at which it excels: performing repetitive tasks repeatedly.

Arrays

At this point, you have dealt with only a few variables in each Java program. In some cases, it's manageable to use individual variables to store information, but what if you had 20 items of related information to track? You could create 20 different variables and set up their initial values, but that approach becomes progressively more cumbersome as you deal with larger amounts of information. What if there were 100 items, or even 1,000?

Arrays are a way to store a list of items that have the same primitive data type, the same class, or a common parent class. Each item on the list goes into its own numbered slot so that you can easily access the information.

Arrays can contain any type of information that is stored in a variable, but after the array is created, you can use it for that information type only. For example, you can have an array of integers, an array of `String` objects, or an array of arrays, but you can't have an array that contains both `String` objects and the primitive type integers.

There is one way around this prohibition: An array can hold a class and any of its subclasses. So an array of the `Object` class could contain any object in Java, including the classes that represent the same values as primitive types.

Java implements arrays differently than other languages—as objects treated like other objects.

To create an array in Java, you must do the following:

1. Declare a variable to hold the array.
2. Create a new array object and assign it to the array variable.
3. Store information in that array.

Declaring Array Variables

The first step in array creation is to declare a variable that will hold the array. Array variables indicate the object or data type that the array will hold and the array's name. To differentiate from regular variable declarations, a pair of empty brackets [] is added to the object or data type, or to the variable name.

The following statements are examples of array variable declarations:

```
String[] requests;
Point[] targets;
float[] donations;
```

You also can declare an array by putting the brackets after the variable name instead of the information type, as in the following statements:

```
String requests[];
Point targets[];
float donations[];
```

Note

The choice of which style to use is a matter of personal preference. The sample programs in this book place the brackets after the information type rather than the variable name, which is the more popular convention among Java programmers.

Creating Array Objects

After you declare the array variable, the next step is to create an array object and assign it to that variable. To do this:

- Use the `new` operator.
- Initialize the contents of the array directly.

Because arrays are objects in Java, you can use the `new` operator to create a new instance of an array, as in the following statement:

Click here to view code image

```
String[] players = new String[10];
```

This statement creates a new array of strings with 10 slots that can contain `String` objects. When you create an array object by using `new`, you must indicate how many slots the array will hold. This statement does not put actual `String` objects in the slots; you must do that later.

Array objects can contain primitive types, such as integers or Booleans, just as they can contain objects:

```
int[] temps = new int[99];
```

When you create an array object using `new`, all its slots automatically are given an initial value (0 for numeric arrays, `false` for Booleans, '\0' for character arrays, and `null` for objects).

Note

The Java keyword `null` refers to a `null` object (and can be used for any object reference). It is not equivalent to 0 or the '\0' character as the `NULL` constant is in C.

Because each object in an array of objects has a `null` reference when created, you must assign an object to each array element before using it.

The following example creates an array of three `Integer` objects and then assigns each element an object:

Click here to view code image

```
Integer[] series = new Integer[3];
series[0] = new Integer(10);
series[1] = new Integer(3);
series[2] = new Integer(5);
```

You can create and initialize an array at the same time by enclosing the array's elements inside braces, separated by commas:

Click here to view code image

```
Point[] markup = { new Point(1,5), new Point(3,3), new Point(2,3)
};
```

Each of the elements inside the braces must be the same type as the variable that holds the array. When you create an array with initial values in this manner, the array is the same size as the number of elements you include

within the braces. The preceding example creates an array of `Point` objects named `markup` that contains three elements.

Because `String` objects can be created and initialized without the `new` operator, you can do the same when creating an array of strings:

Click here to view code image

```
String[] titles = { "Mr.", "Mrs.", "Ms.", "Miss", "Dr." };
```

The preceding statement creates a five-element array of `String` objects named `titles`.

All arrays have an instance variable named `length` that holds a count of the number of elements in the array. Extending the preceding example, the variable `titles.length` contains the value 5.

The first element of an array has a subscript of 0 rather than 1, so an array with five elements has array slots accessed using subscripts 0 through 4.

Accessing Array Elements

After you have an array with initial values, you can retrieve, change, and test the values in each slot of that array. The value in a slot is accessed using the array name followed by a subscript enclosed in square brackets. This name and subscript can be put into expressions, as in the following:

```
testScore[40] = 920;
```

This statement sets the 41st element of the `testScore` array to a value of 920, since element numbering begins at 0. The `testScore` part of this expression is a variable holding an array object, although it also can be an expression that results in an array. The subscript expression specifies the slot to access within the array.

All array subscripts are checked to make sure that they are inside the array's boundaries as specified when the array was created. In Java, it is impossible to access or assign a value to an array slot outside the array's boundaries. This avoids the problems that result from overrunning the bounds of an array in other languages. Note the following two statements:

Click here to view code image

```
float[] rating = new float[20];
rating[20] = 3.22F;
```

Typing these statements into NetBeans would produce an error because the `rating` array does not have a slot numbered 20; it has 20 slots that begin at 0 and end at 19. The Java compiler would fail with an `ArrayIndexOutOfBoundsException` error.

The Java Virtual Machine (JVM) also notes an error if the array subscript is calculated when the program is running and the subscript is outside the array's boundaries. You learn more about errors, which are called exceptions, on Day 7, "Exceptions and Threads."

One way to avoid accidentally overrunning the end of an array in your programs is to use the `length` instance variable. The following statement displays the number of elements in the `rating` array:

Click here to view code image

```
System.out.println("Elements: " + rating.length);
```

Changing Array Elements

As you saw in the previous examples, you can assign a value to a specific slot in an array by putting an assignment statement after the array name and subscript, as in the following:

```
temperature[4] = 85;

day[0] = "Sunday";

manager[2] = manager[0];
```

It's important to remember that an array of objects in Java is an array of references to those objects. When you assign a value to a slot in that kind of array, you are creating a reference to that object. When you move around values inside arrays, you are reassigning the reference rather than copying a value from one slot to another. Arrays of a primitive data type, such as `int` and `float`, do copy the values from one slot to another, as do elements of a `String` array, even though they are objects.

Arrays are simple to create and modify, and they provide an enormous amount of functionality in Java. The HalfDollars application, shown in Listing 4.1, creates, initializes, and displays elements of three arrays. Create a new empty Java file in NetBeans called HalfDollars in the `com.java21days` package, and enter the listing's source code.

LISTING 4.1 The Full Text of `HalfDollars.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class HalfDollars {
 4:     public static void main(String[] arguments) {
 5:         int[] denver = { 1_700_000, 4_600_000, 2_100_000 };
 6:         int[] philadelphia = new int[denver.length];
 7:         int[] total = new int[denver.length];
 8:         int average;
 9:
10:         philadelphia[0] = 1_800_000;
11:         philadelphia[1] = 5_000_000;
12:         philadelphia[2] = 2_500_000;
13:
14:         total[0] = denver[0] + philadelphia[0];
15:         total[1] = denver[1] + philadelphia[1];
16:         total[2] = denver[2] + philadelphia[2];
17:         average = (total[0] + total[1] + total[2]) / 3;
18:
19:         System.out.print("2012 production: ");
20:         System.out.format("%,d%n", total[0]);
21:         System.out.print("2013 production: ");
22:         System.out.format("%,d%n", total[1]);
23:         System.out.print("2014 production: ");
24:         System.out.format("%,d%n", total[2]);
25:         System.out.print("Average production: ");
26:         System.out.format("%,d%n", average);
27:     }
28: }
```

The HalfDollars application uses three integer arrays to store production totals for U.S. half-dollar coins produced at the Denver and Philadelphia mints. When you run the program, it displays the output shown in Figure 4.1.

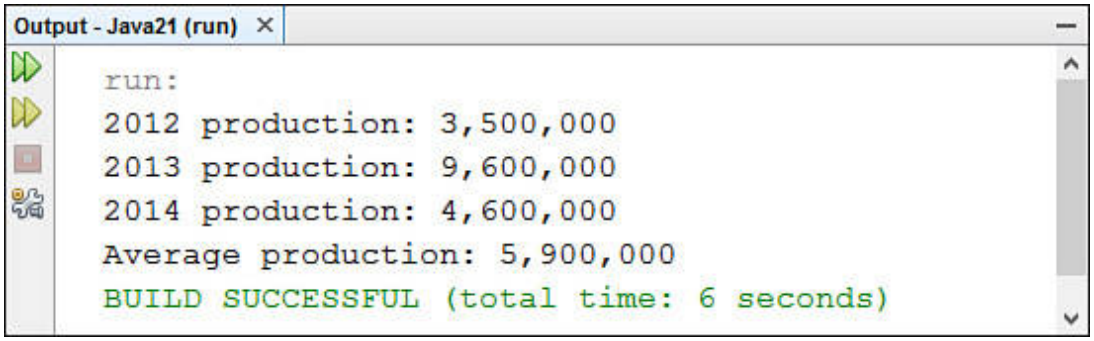


FIGURE 4.1 Displaying the contents of a `String` array.

The class created here, `HalfDollars`, has three instance variables that hold arrays of integers.

The first, which is named `denver`, is declared and initialized on line 5 to contain three integers: 1_700_000 in element 0, 4_600_000 in element 1, and 2_100_000 in element 2. These figures are the total half-dollar production at the Denver mint for three years. The integers use an underscore character _ after every three digits to make the numbers more human-readable. The compiler ignores the underscores.

The second and third instance variables, `philadelphia` and `total`, are declared in lines 6 and 7. The `philadelphia` array contains the production totals for the Philadelphia mint, and `total` is used to store the overall production totals.

No initial values are assigned to the slots of the `philadelphia` and `total` arrays in lines 6 and 7. For this reason, each element is given the default value for integers: 0.

The `denver.length` variable is used to give both of these arrays the same number of slots as the `denver` array. Every array contains a `length` variable that you can use to keep track of the number of elements it contains.

The rest of the `main()` method of this application does the following:

- Line 8 creates an integer variable called `average`.
- Lines 10–12 assign new values to the three elements of the `philadelphia` array.
- Lines 14–16 assign new values to the elements of the `total` array. In line 14, `total` element 0 is given the sum of `denver` element 0 and `philadelphia` element 0. Similar expressions are used in lines 15 and 16.
- Line 17 sets the value of the `average` variable to the average of the three `total` elements. Because `average` and the three `total` elements are integers, the average is expressed as an integer rather than a floating-point number.
- Lines 19–26 display the values stored in the `total` array and the `average` variable, using the `System.out.format()` method to display the numeric values in a more readable form using commas.

This application handles arrays inefficiently. The statements are almost identical, except for the subscripts that indicate the array element to which

you are referring. If the HalfDollars application were being used to track 100 years of production totals instead of three, this approach would require a lot of redundant statements.

When dealing with arrays, you can use loops to cycle through an array's elements instead of dealing with each element individually. This makes the code a lot shorter and easier to read. When you learn about loops later today, you see a rewrite of the current example.

Multidimensional Arrays

Arrays can be multidimensional, containing more than one subscript to store information in multiple dimensions.

A common use of a multidimensional array is to represent the data in an (x,y) grid of array elements.

Java supports this by enabling an array to hold arrays as each of its elements. Those arrays can also contain arrays, and so on, for as many dimensions as needed.

For example, consider a program that needs to accomplish the following tasks:

- Record an integer value each day for a year.
- Organize those values by week.

One way to organize this data is to create a 53-element array in which each element contains a 7-element array:

Click here to view code image

```
int[][] dayValue = new int[53][7];
```

This array of arrays contains a total of 371 integers, enough room for each day of the year (plus a few extra). You could set the value for the first day of the 10th week with the following statement:

```
dayValue[9][0] = 14200;
```

Remember that array indexes start at 0 instead of 1, so the 10th week is at element 9 and the first day at element 0.

You can use the `length` instance variable with these arrays as you would any other. The following statement contains a three-dimensional array of integers and displays the number of elements in each dimension:

Click here to view code image

```
int[][][] cen = new int[100][52][7];
System.out.println("Elements in 1st dimension: " + cen.length);
System.out.println("Elements in 2nd dimension: " +
cen[0].length);
System.out.println("Elements in 3rd dimension: " + cen[0]
[0].length);
```

Block Statements

Statements in Java are grouped into blocks. The beginning and ending boundaries of a block are noted with brace characters—an opening brace { for the beginning and a closing brace } for the ending.

You have used blocks to hold the variables and methods in a class definition and define statements that belong in a method.

Blocks also are called block statements because an entire block can be used anywhere a single statement could be used. Each statement inside the block then is executed from top to bottom.

You can put blocks inside other blocks, just as you do when you put a method inside a class definition.

An important thing to note about using a block is that it creates a scope for the local variables created inside the block. Scope is the part of a program where a variable exists and can be used. If you try to use a variable outside its scope, an error occurs.

In Java, the scope of a variable is the block in which it was created. When you can declare and use local variables inside a block, those variables cease to exist after the block is finished executing. For example, the following method contains a block:

```
void testBlock() {
    int x = 10;
    { // start of block
        int y = 40;
        y = y + x;
    } // end of block
}
```

Two variables are defined in this method: `x` and `y`. The scope of the `y` variable is the block it's in, which is marked by the comments `// start of block` and `// end of block`. The variable can be used only within

that block. An error would result if you tried to use the `y` variable in another part of the method.

The `x` variable was created inside the method but outside the inner block, so it can be used anywhere in the method. You can modify the value of `x` anywhere within the method.

Block statements are used in class and method definitions and the logic and looping structures you learn about next. The way the preceding example uses the inner block is not common.

If Conditionals

A key aspect of any programming language is how it enables a program to make decisions. This is handled through a type of statement called a *conditional*, a statement executed only if a specific condition is met.

The most basic conditional in Java is `if`. The `if` conditional uses a Boolean expression to decide whether a statement should be executed. If the expression produces a `true` value, the statement is executed.

Here's a simple example that displays the message "Not enough arguments" only if the value of an instance variable is less than 3:

Click here to view code image

```
if (arguments.length < 3) {
    System.out.println("Not enough arguments");
    System.exit(-1);
}
```

If you want something else to happen when an `if` expression is not `true`, you can use the `else` keyword. The following example uses both `if` and `else`:

```
String server;
int duration;
if (arguments.length < 1) {
    server = "localhost";
} else {
    server = arguments[0];
}
```

The `if` conditional executes different statements based on the result of a single Boolean test.

Note

A difference between `if` conditionals in Java and those in other languages is that Java conditionals produce only Boolean values (`true` or `false`). In C and C++, the test can return an integer.

Using `if`, you can include only a single statement as the code to execute if the test expression is true and another statement if the expression is false.

However, as you learned earlier today, a block can appear anywhere in Java that a single statement can appear. If you want to do more than one thing as a result of an `if` statement, you can enclose those statements inside a block. Note the following code, which was used on Day 1, "Getting Started with Java":

```
int speed;
String status;
float temperature = -60;

if (temperature < -80) {
    status = "returning home";
    speed = 5;
}
```

The `if` statement in this example contains the test expression `temperature < -80`. If the `temperature` variable contains a value less than –80, the block statement is executed, and two things occur:

- The `status` variable is given the value "returning home."
- The `speed` variable is set to 5.

If the `temperature` variable is equal to or greater than –80, the entire block is skipped, so nothing happens.

All `if` and `else` statements use Boolean tests to determine whether statements are executed. You can use a `boolean` variable itself for this test, as in the following:

```
String status;
boolean outOfGas = true;
if (outOfGas) {
    status = "inactive";
}
```

The preceding example uses a `boolean` variable called `outOfGas`. It functions exactly like the following:

```
if (outOfGas == true) {
    status = "inactive";
}
```

Switch Conditionals

A common programming practice is to test a variable against a value, and if it doesn't match, test it again against a different value, and so on.

This approach can become unwieldy if you're using only `if` statements, depending on how many different values you have to test. For example, you might end up with a set of `if` statements something like the following:

Click here to view code image

```
if (operation == '+')
    add(object1, object2);
else if (operation == '-')
    subtract(object1, object2);
else if (operation == '*')
    multiply(object1, object2);
else if (operation == '/')
    divide(object1, object2);
```

This use of `if` statements is called a *nested `if` statement* because each `else` statement contains another `if` until all possible tests have been made.

A better way to handle this situation in Java is by grouping actions with the `switch` statement. The following example demonstrates `switch` usage:

Click here to view code image

```
char grade = 'D';
switch (grade) {
    case 'A':
        System.out.println("Great job!");
        break;
    case 'B':
        System.out.println("Good job!");
        break;
    case 'C':
        System.out.println("You can do better!");
        break;
    default:
        System.out.println("Consider cheating!");
}
```

A `switch` statement is built on a test variable. In the preceding example, the variable is the value of the `grade` variable, which holds a `char` value.

The test variable can be the primitive types `byte`, `char`, `short`, or `int` or the class `String`. The following code uses the value of a `String` object named `command` to decide which method to call:

Click here to view code image

```
String command = "close";
switch (command) {
    case "open":
        openFile();
        break;
    case "close":
        closeFile();
        break;
    default:
        System.out.println("Invalid command");
    }
}
```

The test variable is compared in turn with each `case` value. If a match is found, the statement or statements after the test are executed.

If no match is found, the `default` statement or statements are executed. Providing a `default` statement is optional. If it is omitted and there is no match for any of the `case` statements, the `switch` statement might complete without executing anything.

The test cases in a `switch` statement are limited to primitive types that can be cast to an `int`, such as `char` or strings. You cannot use larger primitive types such as `long` or `float` or test for any relationship other than equality.

The following is a revision of the nested `if` example shown previously. It has been rewritten as a `switch` statement:

Click here to view code image

```
switch (operation) {
    case '+':
        add(object1, object2);
        break;
    case '-':
        subtract(object1, object2);
        break;
    case '*':
        multiply(object1, object2);
        break;
    case '/':
```

```
            divide(object1, object2);
            break;
    }
```

After each `case`, you can include a single result statement or as many as you need. Unlike `if` statements, multiple statements don't require a block statement.

The `break` statement included with each `case` section determines when to stop executing statements in response to a matching `case`. Suppose a `case` section has no `break` statement. After a match is made, the statements for that match and all the statements further down the `switch` are executed until a `break` or the end of the switch is found.

In some situations, this might be exactly what you want to do. Otherwise, you should include `break` statements to ensure that only the right code is executed. The `break` statement, which you use again later in the section "Breaking Out of Loops," stops execution at the current point. Then it jumps to the statement after the closing brace that ends the `switch` statement.

One handy use of falling through without a `break` occurs when multiple values need to execute the same statements. To accomplish this task, you can use multiple `case` lines with no result; the `switch` executes the first statement it finds.

For example, in the following `switch` statement, the string "x is an even number" is printed if `x` has a value of 2, 4, 6, or 8. All other values of `x` cause the string "x is an odd number" to be printed.

Click here to view code image

```
int x = 5;
switch (x) {
    case 2:
    case 4:
    case 6:
    case 8:
        System.out.println("x is an even number");
        break;
    default:
        System.out.println("x is an odd number");
}
```

The next project for today, the DayCounter application in Listing 4.2, takes a month and a year as arguments and displays the number of days in that month. A `switch` statement, `if` statements, and `else` statements are used. Create

this application in NetBeans as an empty Java file in the `com.java21days` package.

LISTING 4.2 The Full Text of `DayCounter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class DayCounter {
 4:     public static void main(String[] arguments) {
 5:         int yearIn = 2016;
 6:         int monthIn = 1;
 7:         if (arguments.length > 0)
 8:             monthIn = Integer.parseInt(arguments[0]);
 9:         if (arguments.length > 1)
10:             yearIn = Integer.parseInt(arguments[1]);
11:         System.out.println(monthIn + "/" + yearIn + " has "
12:             + countDays(monthIn, yearIn) + " days.");
13:     }
14:
15:     static int countDays(int month, int year) {
16:         int count = -1;
17:         switch (month) {
18:             case 1:
19:             case 3:
20:             case 5:
21:             case 7:
22:             case 8:
23:             case 10:
24:             case 12:
25:                 count = 31;
26:                 break;
27:             case 4:
28:             case 6:
29:             case 9:
30:             case 11:
31:                 count = 30;
32:                 break;
33:             case 2:
34:                 if (year % 4 == 0)
35:                     count = 29;
36:                 else
37:                     count = 28;
38:                 if ((year % 100 == 0) & (year % 400 != 0))
39:                     count = 28;
40:         }
41:         return count;
```

```
42:       }
43: }
```

This application uses command-line arguments to specify the month and year to check. The first argument is the month, which should be expressed as a number from 1 to 12. The second argument is the year, which should be expressed as a full four-digit year.

If the application is run without setting the arguments, it uses 1 as the month and 12 as the year, displaying the output in Figure 4.2.

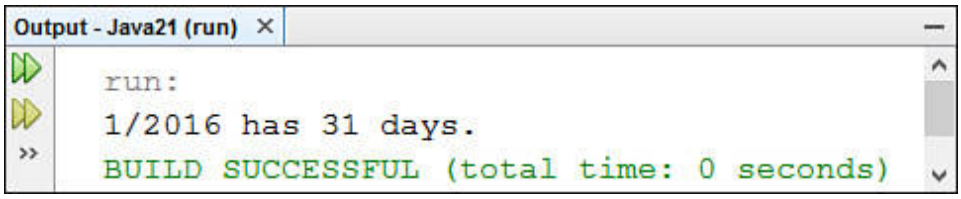


FIGURE 4.2 Using `switch-case` to handle numerous conditionals.

To set command-line arguments in NetBeans, choose Run, Set Project Configuration, Customize. The Project Properties dialog opens, as shown in Figure 4.3.

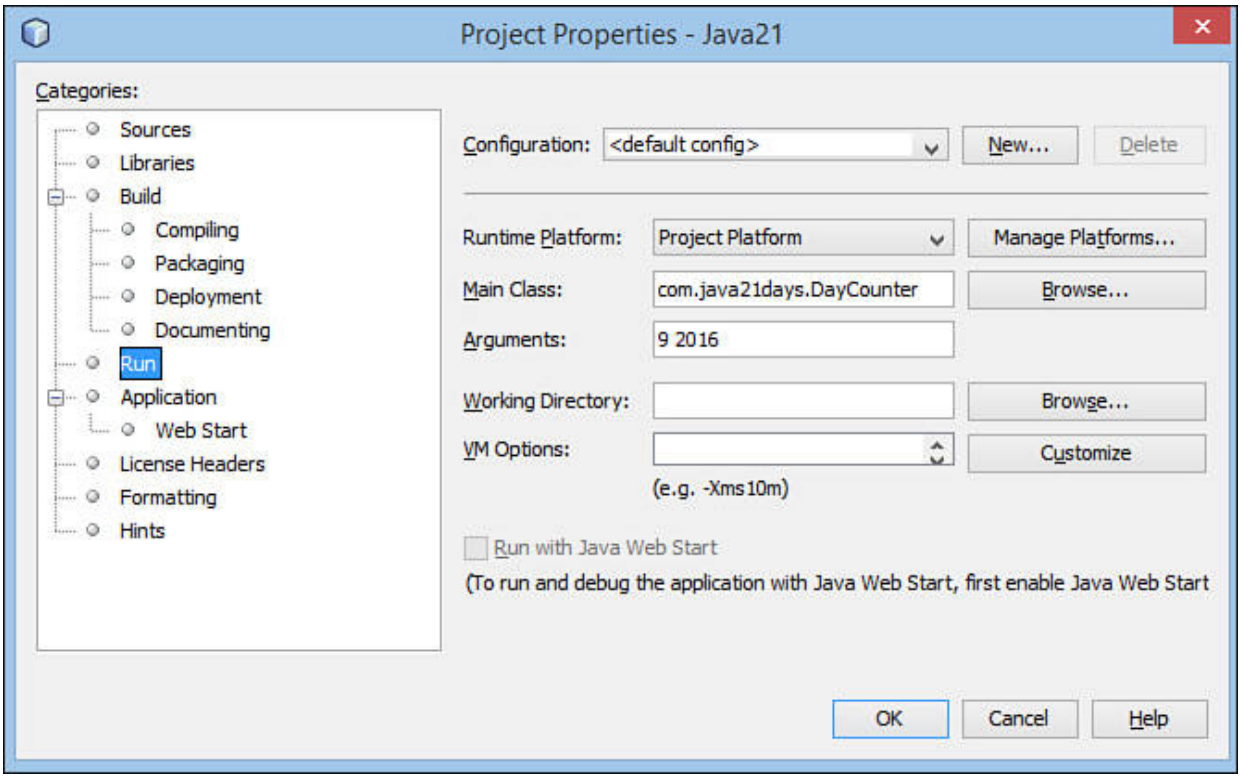


FIGURE 4.3 Setting command-line arguments for an application in NetBeans.

In the Main Class field, enter the name of the class that contains the `main()` method that will be run: `com.java21days.DayCounter`.

In the Arguments field, enter the command-line arguments separated by spaces, such as `9 2016`. Click OK to save this configuration.

To run the application with these arguments in NetBeans, choose Run, Run Project (instead of Run, Run File). When run with 9 and 2016 as arguments, the output is that shown in Figure 4.4.

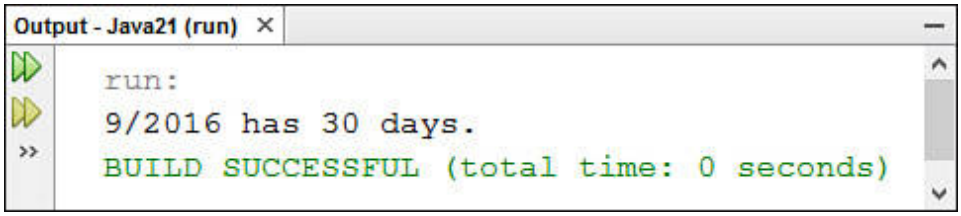


FIGURE 4.4 Using `switch-case` to handle numerous conditionals.

The `DayCounter` application uses a `switch` statement to count the days in a month. This statement is part of the `countDays()` method in lines 15–42 of Listing 4.2.

The `countDays()` method has two `int` arguments: `month` and `year`. The number of days is stored in the `count` variable, which is given an initial value of –1 that is replaced by the correct count later.

The `switch` statement that begins on line 17 uses `month` as its conditional value.

The number of days in a month is easy to determine for 11 months of the year. January, March, May, July, August, October, and December have 31 days. April, June, September, and November have 30 days.

The count for these 11 months is handled in lines 18–32 of Listing 4.2. Months are numbered from 1 (January) to 12 (December), as you would expect. When one of the `case` statements has the same value as `month`, every statement after that is executed until `break` or the end of the `switch` statement is reached.

February is more complex and is handled in lines 33–39. Every leap year has 29 days in February, whereas other years have 28. A leap year must meet either of the following conditions:

- The year must be evenly divisible by 4 and not evenly divisible by 100.
- The year must be evenly divisible by 400.

As you learned on Day 2, "The ABCs of Programming," the modulus operator % returns the remainder of a division operation. This is used with several `if-else` statements to determine how many days there are in February, depending on what year it is.

The `if-else` statement in lines 34–37 sets `count` to 29 when the year is evenly divisible by 4 and sets it to 28 otherwise.

The `if` statement in lines 38–39 uses the `&` operator to combine two conditional expressions: `year % 100 == 0` and `year % 400 != 0`. If both these conditions are true, `count` is set to 28.

The `countDays` method ends by returning the value of `count` in line 41.

When you run the DayCounter application, the `main()` method in lines 4–13 is executed.

In all Java applications, command-line arguments are stored in an array of `String` objects. This array is called `arguments` in DayCounter. The first command-line argument is stored in `argument[0]`, the second in `argument[1]`, and upward until all arguments have been stored. If the application is run with no arguments, the array is created with no elements.

Lines 5 and 6 create `yearIn` and `monthIn`, two integer variables to store the year and month that should be checked.

The `if` statement in line 7 uses `arguments.length` to make sure that the `arguments` array has at least one element. If it does, line 8 is executed.

Line 10 calls `parseInt()`, a class method of the `Integer` class, with `arguments[0]` as an argument. This method takes a `String` object as an argument, and if the string could be a valid integer, it returns that value as an `int`. This converted value is stored in `monthIn`. A similar thing happens in line 10: `parseInt()` is called with `arguments[1]`, and this is used to set `yearIn`.

The program's output is displayed in lines 11–12. As part of the output, the `countDays()` method is called with `monthIn` and `yearIn`, and the value returned by this method is displayed.

Note

At this point, you might want to know how to collect input from a user in a program rather than using command-line arguments to receive it. There isn't a method comparable to `System.out.println()` that receives input. Instead, you must learn a bit more about Java's input and output classes before you can receive input in a program without a graphical user interface. This topic is covered during Day 15, "Working with Input and Output."

The Ternary Operator

An alternative to using the `if` and `else` keywords in a conditional statement is to use the ternary operator, also called the conditional operator. This operator is ternary because it has three operands (the word "ternary" refers to anything with three parts).

This operator is an expression, meaning that it returns a value—unlike the more general `if`, which can result in only a statement or block being executed. The operator is most useful for short or simple conditionals and takes the following form:

Click here to view code image

```
test ? trueResult : falseResult;
```

The *test* is an expression that returns `true` or `false`, just like the test in the `if` statement. If the *test* is `true`, the conditional operator returns the value of *`trueResult`*. If the *test* is `false`, the conditional operator returns the value of *`falseResult`*. For example, the following conditional tests the values of `myScore` and `yourScore` and sets the variable `ourBestScore` equal to one of them:

Click here to view code image

```
int ourBestScore = myScore > yourScore ? myScore : yourScore;
```

In this statement, the larger value of `myScore` and `yourScore` is copied to `ourBestScore`.

This use of the ternary operator is equivalent to the following `if-else` code:

```
int ourBestScore;
if (myScore > yourScore) {
```

```
    ourBestScore = myScore;
} else {
    ourBestScore = yourScore;
}
```

The ternary operator has low precedence. Usually it is evaluated only after all its subexpressions have been evaluated. The only operators lower in precedence are the assignment operators. For a refresher on operator precedence, refer to Table 2.6 in Day 2.

Note

The ternary operator is of primary benefit to experienced programmers creating complex expressions. Because its functionality is duplicated in simpler use of `if-else` statements, there's no need to use this operator while you're beginning to learn the language. The main reason it's introduced in this book is because you'll encounter it in the source code of other Java programmers.

For Loops

A `for` loop is used to repeat a statement until a condition is met. Although `for` loops frequently are used for simple iteration in which a statement is repeated a certain number of times, `for` loops can be used for just about any kind of loop.

The `for` loop in Java has the following structure:

Click here to view code image

```
for (initialization; test; increment) {
    statement;
}
```

The start of the `for` loop has three parts:

- The *`initialization`* is an expression that initializes the start of the loop. If you have a loop index, this expression might declare and initialize it, such as `int i = 0`. Variables that you declare in this part of the `for` loop are local to the loop itself. They cease to exist after the loop is finished executing. You can initialize more than one variable in this section by separating each expression with a comma. The statement `int i = 0, j = 10` in this section would declare the variables `i` and `j`, and both would be local to the loop.

- The *test* is the test that occurs before each pass of the loop. The test must be a Boolean expression or a function that returns a `boolean` value, such as `i < 10`. If the test is `true`, the loop executes. When the test is `false`, the loop stops executing.
- The *increment* is any expression or method call. Commonly, the increment is used to change the value of the loop index to bring the state of the loop closer to returning `false` and stopping the loop. The increment takes place after each pass of the loop. Similar to the *initialization* section, you can put more than one expression in this section by separating each expression with a comma.

The *statement* part of the `for` loop is the statement that is executed each time the loop iterates. As with `if`, you can include either a single statement or a block statement. The previous example used a block because that is more common. The following example is a `for` loop that sets all slots of a `String` array to the value "Mr.":

Click here to view code image

```
String[] salutation = new String[10];
int i; // the loop index variable
for (i = 0; i < salutation.length; i++) {
    salutation[i] = "Mr.";
}
```

In this example, the variable `i` serves as a loop index; it counts the number of times the loop has been executed. Before each trip through the loop, the index value is compared with `salutation.length`, the number of elements in the `salutation` array. When the index is equal to or greater than `salutation.length`, the loop is exited.

The final element of the `for` statement is `i++`. This causes the loop index to increment by 1 each time the loop is executed. Without this statement, the loop would never stop.

The statement inside the loop sets an element of the `salutation` array equal to "Mr." The loop index is used to determine which element is modified.

Any part of the `for` loop can be an empty statement; in other words, you can include a semicolon with no expression or statement, and that part of the `for` loop is ignored. Note that if you do use an empty statement in your `for` loop,

you might have to initialize or increment any loop variables or loop indexes yourself elsewhere in the program.

You also can have an empty statement as the body of your `for` loop if everything you want to do is in the first line of that loop. For example, the following `for` loop finds the first prime number higher than 4,000. (It assumes the existence of a method called `notPrime()` that returns a Boolean value to indicate when `i` is not prime.)

Click here to view code image

```
for (i = 4001; notPrime(i); i += 2);
```

The semicolon at the end of the `for` statement indicates that the loop has no statements in its body.

A common mistake in `for` loops is to accidentally put a semicolon at the end of the line that includes the `for` statement:

Click here to view code image

```
int x = 1;
for (i = 0; i < 10; i++);
    x = x * i; // this line is not inside the loop!
```

In this example, the semicolon outside the parentheses in the `for` statement ends the loop without executing `x = x * i` as part of the loop. The `x = x * i` line is executed only once because it is outside the `for` loop. Be careful not to make this mistake in your Java programs.

The next project you undertake is a rewrite of the HalfDollar application that uses `for` loops to remove redundant code.

The original application works with an array that is only three elements long. The new version shown in Listing 4.3, called HalfLooper, is shorter and more flexible and returns the same output. Create an empty Java file with that class name and the package name `com.java21days` in NetBeans.

LISTING 4.3 The Full Text of `HalfLooper.java`

Click here to view code image

```
1: package com.java21days;
2:
3: class HalfLooper {
4:     public static void main(String[] arguments) {
5:         int[] denver = { 1_700_000, 4_600_000, 2_100_000 };
```

```
 6:          int[] philadelphia = { 1_800_000, 5_000_000,
2_500_000 };
 7:          int[] total = new int[denver.length];
 8:          int sum = 0;
 9:
10:          for (int i = 0; i < denver.length; i++) {
11:              total[i] = denver[i] + philadelphia[i];
12:              System.out.format((i + 2012) + " production:
%,d%n",
13:                  total[i]);
14:              sum += total[i];
15:          }
16:
17:          System.out.format("Average production: %,d%n",
18:              (sum / denver.length));
19:      }
20: }
```

The output is the same as for the HalfDollars application in Figure 4.1.

Instead of going through the elements of the three arrays one by one, this example uses a `for` loop. The following things take place in the loop, which is contained in lines 10–15:

- **Line 10**—The loop is created with an `int` variable called `i` as the index. The index increments by 1 for each pass through the loop and stops when `i` is equal to or greater than `denver.length`, the total number of elements in the `denver` array.
- **Lines 11–12**—The value of one of the `total` elements is set using the loop index and then is displayed with some text identifying the year.
- **Line 14**—The value of a `total` element is added to the `sum` variable, which is used to calculate the average yearly production.

Using a more general-purpose loop to iterate over an array enables you to use the program with arrays of different sizes and still have it assign correct values to the elements of the total array and display those values.

Note

Java also includes a `for` loop that can be used to iterate through all the elements of data structures, such as array lists, linked lists, hash maps, and other collections. This loop is covered along with those structures on Day 8, "Data Structures."

While and Do Loops

The remaining types of loops are `while` and `do`, which also enable a block of Java code to be executed repeatedly until a specific condition is met.

While Loops

The `while` loop repeats a statement for as long as a particular condition remains `true`. Here's an example:

Click here to view code image

```
while (i < 13) {
    x = x * i++; // the body of the loop
}
```

The condition that accompanies the `while` keyword is a Boolean expression—`i < 13` in the preceding example. If the expression returns `true`, the `while` loop executes the body of the loop and then tests the condition again. This process repeats until the condition is `false`.

Although the preceding loop uses opening and closing braces to form a block statement, the braces are unneeded because the loop contains only one statement: `x = x * i++`. Using the braces does not create any problems, though, and the braces will be required if you add another statement inside the loop later.

The ArrayCopier application in Listing 4.4 uses a `while` loop to copy the elements of an array of integers (`array1`) to an array of `float` variables (`array2`), casting each element to a `float` as it goes. The one catch is that if any of the elements in the first array is 1, the loop immediately exits at that point.

Create an empty Java file in NetBeans with the class name `ArrayCopier` and package `com.java21days`. Enter Listing 4.4 as the source code.

LISTING 4.4 The Full Text of `ArrayCopier.java`

Click here to view code image

```
1: package com.java21days;
2:
3: class ArrayCopier {
4:     public static void main(String[] arguments) {
5:         int[] array1 = { 7, 4, 8, 1, 4, 1, 4 };
```

```
 6:          float[] array2 = new float[array1.length];
 7:
 8:          System.out.print("array1: [ ");
 9:          for (int i = 0; i < array1.length; i++) {
10:              System.out.print(array1[i] + " ");
11:          }
12:          System.out.println("]");
13:
14:          System.out.print("array2: [ ");
15:          int count = 0;
16:          while ( count < array1.length && array1[count] != 1)
{
17:              array2[count] = (float) array1[count];
18:              System.out.print(array2[count++] + " ");
19:          }
20:          System.out.println("]");
21:      }
22: }
```

The output is shown in Figure 4.5.

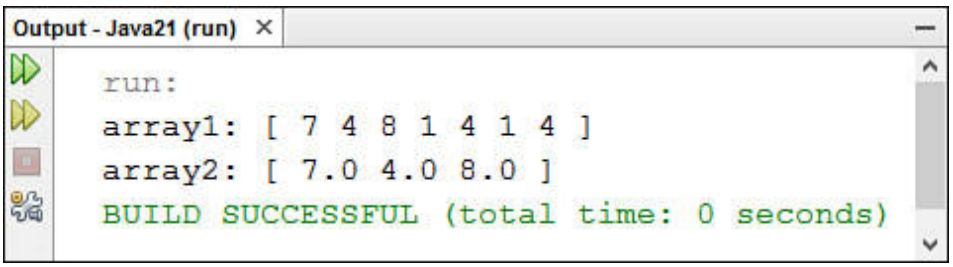


FIGURE 4.5 Using a `while` loop to examine an array.

Here is what's going on in the `main()` method:

- Lines 5 and 7 declare the arrays. `array1` is an array of integers, which are initialized to some suitable numbers. `array2` is an array of floating-point numbers the same length as `array1`.
- Lines 8–12 iterate through `array1` using a `for` loop to print its values.
- Lines 14–20 assign the values of `array2` (converting the numbers to floating-point numbers along the array) and print them. You start with a `count` variable, which keeps track of the array index elements. The test in the `while` loop keeps track of the two conditions for exiting the loop, where those two conditions are running out of elements in `array1` or encountering a 1 in `array1`.

  You can use the logical conditional `&&` operator to keep track of the test; remember that `&&` makes sure that both conditions are `true` before the

entire expression is `true`. If either one is `false`, the expression returns `false`, and the loop exits.

The program's output shows that the first four elements in `array1` were copied to `array2`, but a 1 in the middle stopped the loop from going any further. Without the 1, `array2` should end up with all the same elements as `array1`. If the `while` loop's test initially is `false` the first time it is tested (for example, if the first element in that first array is 1), the body of the `while` loop will never be executed. If you need to execute the loop at least once, you can do one of two things:

- Duplicate the body of the loop outside the `while` loop.
- Use a `do` loop (which is described in the following section).

The `do` loop is considered the better solution.

Do-While Loops

The `do` loop is like a `while` loop, with one major difference—the place in the loop where the condition is tested.

A `while` loop tests the condition before looping, so if the condition is `false` the first time it is tested, the body of the loop never executes.

A `do` loop executes the body of the loop at least once before testing the condition. So if the condition is `false` the first time it is tested, the body of the loop already will have executed once.

The following example uses a `do` loop to keep doubling the value of a `long` integer until it is larger than 3 trillion:

Click here to view code image

```
long i = 1;
do {
    i *= 2;
    System.out.print(i + " ");
} while (i < 3_000_000_000_000L);
```

The body of the loop is executed once before the test condition, `i < 3_000_000_000_000L`, is evaluated. Then, if the test evaluates as `true`, the loop runs again. If it is `false`, the loop exits. Keep in mind that the body of the loop executes at least once with `do` loops.

The `for`, `while`, and `do` loops all accomplish the same purpose in slightly different ways. When writing your own code, you may have trouble deciding

which one to use. There's often no wrong answer. Whether you use a `for`, `while`, or `do` loop is largely a matter of preference.

Breaking Out of Loops

All loops end when a tested condition is met. There might be times when something occurs during execution of a loop, and you want to exit the loop early. In that case, you can use the `break` and `continue` keywords.

You already have seen `break` as part of the `switch` statement; `break` stops execution of the `switch` statement, and the program continues. The `break` keyword, when used with a loop, does the same thing—it immediately halts execution of the current loop. If you have nested loops within loops, execution picks up with the next outer loop. Otherwise, the program continues executing the next statement after the loop.

For example, recall the `while` loop from the ArrayCopier application in Listing 4.4. It copied elements from an integer array into an array of floating-point numbers until either the end of the array or a 1 was reached. You can test for the latter case inside the body of the `while` loop and then use `break` to exit the loop:

Click here to view code image

```
int count = 0;
while (count < array1.length) {
    if (array1[count] == 1) {
        break;
    }
    array2[count] = (float) array2[count++];
}
```

The `continue` keyword starts the loop over at the next iteration. For `do` and `while` loops, this means that the execution of the block statement starts over again; with `for` loops, the increment expression is evaluated, and then the block statement is executed.

The `continue` keyword is useful when you want to make a special case out of elements within a loop. With the previous example of copying one array to another, you could test for whether the current element is equal to 1 and use `continue` to restart the loop after every 1 so that the resulting array never contains 0. Note that because you're skipping elements in the first array, you now have to keep track of two different array counters:

Click here to view code image

```
int count = 0;
int count2 = 0;
while (count++ <= array1.length) {
    if (array1[count] == 1) {
       continue;
    }
    array2[count2++] = (float) array1[count];
}
```

Labeled Loops

Both `break` and `continue` can have an optional label that indicates where to resume execution of the program. Without a label, `break` jumps outside the nearest loop to an enclosing loop or to the next statement outside the loop. The `continue` keyword restarts the loop it is enclosed within. Using `break` and `continue` with a label enables you to use `break` to go to a point outside a nested loop or to use `continue` to go to a loop outside the current loop.

To use a labeled loop, add the label before the initial part of the loop with a colon between the label and the loop. Then, when you use `break` or `continue`, add the name of the label after the keyword itself, as in the following:

Click here to view code image

```
out: for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 50; j++) {
        if (i * j > 400) {
            break out;
        }
    }
}
```

In this code snippet, the label `out` labels the outer loop. Then, inside both the `for` loops, when a particular condition is met, a `break` causes the execution to break out of both loops. Without the label `out`, the `break` statement would exit the inner loop and resume execution with the outer loop.

Labeled loops are used infrequently in Java. There's usually another way to accomplish the same thing.

Summary

Now that you have been introduced to lists, loops, and logic, you can make a computer decide whether to repeatedly display the contents of an array.

You've learned how to declare an array variable, assign an object to it, and access and change elements of the array. With the `if` and `switch` conditional statements, you can branch to different parts of a program based on a Boolean test. You learned about the `for`, `while`, and `do` loops, and you learned that each enables a portion of a program to be repeated until a given condition is met.

It bears repeating: You'll use all three of these features frequently in your Java programs.

You'll use all three of these features frequently in your Java programs.

Q&A

Q I declared a variable inside a block statement for an `if`. When the `if` was done, the definition of that variable vanished. Where did it go?

A In technical terms, block statements form a new lexical scope. This means that if you declare a variable inside a block, it's visible and usable only inside that block. When the block finishes executing, all the variables you declared go away.

It's a good idea to declare most of your variables in the outermost block in which they'll be needed—usually at the top of a block statement. The exception might be simple variables, such as index counters in `for` loops, where declaring them in the first line of the `for` loop is an easy shortcut.

Q Why can't I use `switch` with strings?

A You can. If it isn't working in NetBeans, you must make sure that you have a current version of Java installed and your development environment has been set up to use it.

In NetBeans, to see whether the current project is set up for Java 8, choose File, Project Properties to open the properties dialog. Choose `Libraries` in the Categories pane; then set Java Platform to `JDK 8` if it isn't already. Click OK to save the change and exit the dialog.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What kind of loop is used to execute the statements in the loop at least once before the conditional expression is evaluated?

   A. `do-while`

   B. `for`

   C. `while`

2. Which of the following cannot be used as the test in a `case` statement?

   A. characters

   B. strings

   C. objects

3. Which instance variable of an array is used to find out how big it is?

   A. `size`

   B. `length`

   C. `MAX_VALUE`

Answers

1. A. In a `do-while` loop, the `while` conditional statement appears at the end of the loop. Even if it is initially false, the statements in the loop are executed once.

2. C. It used to be true that strings could not be used as the test, but that is no longer the case.

3. B. The `length` variable is an integer that returns the array's size.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class Cases {
    public static void main(String[] arguments) {
        float x = 9;
        float y = 5;
```

```
            int z = (int)(x / y);
            switch (z) {
                case 1:
                    x = x + 2;
                case 2:
                    x = x + 3;
                default:
                    x = x + 1;
            }
            System.out.println("Value of x: " + x);
        }
    }
```

What will be the value of `x` when it is displayed?

A. 9.0

B. 11.0

C. 15.0

D. The program will not compile.

The answer is available on the book's website at www.java21days.com. Visit the Day 4 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Using the `countDays()` method from the DayCounter application, create an application that displays every date in a given year in a single list from January 1 to December 31.
2. Create a class that takes words for the first 10 numbers ("one" to "ten") and converts them into a single `long` integer. Use a `switch` statement for the conversion and command-line arguments for the words.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 5. Creating Classes and Methods

If you're coming to Java from another programming language, you might be struggling with the meaning of the term *class*. It seems synonymous with the term *program*, but you might be uncertain of the relationship between the two.

In Java, a program is made up of a main class and any other classes needed to support the main class. These support classes include any you might need in the Java Class Library, such as `String`, `Math`, and the like.

Today, the meaning of class is clarified as you create classes and methods, which define the behavior of an object or class. You learn about each of the following:

- The parts of a class
- The creation and use of instance variables
- The creation and use of methods
- The use of the `main()` method in applications
- The creation of overloaded methods
- The creation of constructors

Defining Classes

Because you have created classes during each of the previous days, you should be familiar with the basics of their creation at this point. A class is defined via the `class` keyword and the name of the class, as in the following example:

```
class Ticker {
    // body of the class
}
```

By default, classes inherit from the `Object` class, the superclass of all classes in the Java class hierarchy.

The `extends` keyword is used to indicate the superclass of a class, as in this example, which is defined as a subclass of `Ticker`:

Click here to view code image

```
class SportsTicker extends Ticker {
    // body of the class
}
```

A class that does not use `extends` to identify its superclass has `Object` as its superclass.

Creating Instance and Class Variables

Whenever you create a class, one thing you must do is define behavior that makes the new class different from its superclass.

This behavior is defined by specifying the variables and methods of the new class. In this section, you work with three kinds of variables: instance variables, local variables, and class variables. The subsequent section covers methods.

Defining Instance Variables

On Day 2, "The ABCs of Programming," you learned how to declare and initialize local variables, which are variables inside method definitions.

Instance variables are declared and defined in almost the same manner as local variables. The main difference is their location in the class definition.

Variables are considered instance variables if they are declared outside a method definition and are not modified by the `static` keyword.

By programming custom, most instance variables are defined right after the first line of the class definition, but they could just as easily be defined at the end.

Here's a simple class definition for the class `MarsRobot`, which inherits from the superclass `ScienceRobot`:

Click here to view code image

```
class MarsRobot extends ScienceRobot {
    String status;
    int speed;
    float temperature;
    int power;
}
```

This class definition contains four variables. Because these variables are not defined inside a method, they are instance variables. The variables are as follows:

- `status`—A string indicating the robot's current activity (for example, "exploring" or "returning home")
- `speed`—An integer that indicates the robot's current rate of travel

- `temperature`—A floating-point number that indicates the current temperature of the robot's environment
- `power`—An integer indicating the robot's current battery power

Class Variables

As you learned in previous days, class variables apply to a class as a whole, rather than to a particular object of that class.

Class variables are good for sharing information between different objects of the same class or for keeping track of common information among a set of objects.

The `static` keyword is used in the class declaration to declare a class variable, as in the following example:

Click here to view code image

```
static int SUM;
static final int MAX_OBJECTS = 10;
```

By convention, many Java programmers capitalize the entire names of class variables so that they're distinguished in code from other variables. This is not a requirement of the language, but is a practice that's recommended.

Creating Methods

As you learned on Day 3, "Working with Objects," methods define an object's behavior—anything that happens when the object is created as well as the various tasks the object can perform during its lifetime.

This section introduces method definitions and how methods work. Tomorrow's lesson has more details about more sophisticated things you can do with methods.

Defining Methods

In Java, a method definition has four basic parts:

- The method's name
- A list of parameters
- The type of object or primitive type that the method returns
- The body of the method

The first two parts of the method definition form the method's *signature*.

Note

To keep things simpler today, two optional parts of the method definition have been left out: a modifier, such as `public` or `private`, and the `throws` keyword, which indicates the exceptions a method can throw. You learn about these parts of method definition on Day 6, "Packages, Interfaces, and Other Class Features," and Day 7, "Exceptions and Threads."

In other languages, the name of the method (which might be called a function, subroutine, or procedure) is enough to distinguish it from other methods in the program.

In Java, you can have several methods in the same class with the same name but different signatures. This practice is called method overloading, and you learn more about it later today.

Here's what a basic method definition looks like:

Click here to view code image

```
returnType methodName(type1 arg1, type2 arg2, type3 arg3 ...) {
    // body of method
}
```

The *`returnType`* is the primitive type or class of the value returned by the method. It can be one of the primitive types, such as `int` or `float`, a class name, or `void` if the method does not return a value.

The method's parameter list is a set of variable declarations separated by commas and set inside parentheses. These parameters become local variables in the body of the method, receiving their values when the method is called.

If a method returns an array object, the array brackets can go after either the return type or the closing parenthesis of the parameter list. Because putting the brackets after the return type is easier to read, that approach is used in this book. For instance, the following declares a method that returns an integer array:

Click here to view code image

```
int[] makeRange(int lower, int upper) {
    // body of method
}
```

Inside the body of a method, you can have statements, expressions, method calls on other objects, conditionals, loops, and so on.

Unless a method has been declared with `void` as its return type, the method returns some kind of value when it is completed. This value must be explicitly returned at some exit point inside the method by using the `return` keyword.

Listing 5.1 contains `RangeLister`, a class that defines a `makeRange()` method. This method takes two integers—a lower boundary and an upper boundary—and creates an array that contains all the integers between those two boundaries. The boundaries themselves are included in the array of integers.

Create a new empty Java file in NetBeans for a class called `RangeLister` (package `com.java21days`) and enter the code of Listing 5.1 into it.

LISTING 5.1 The Full Text of `RangeLister.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class RangeLister {
 4:     int[] makeRange(int lower, int upper) {
 5:         int[] range = new int[(upper-lower) + 1];
 6:
 7:         for (int i = 0; i < range.length; i++) {
 8:             range[i] = lower++;
 9:         }
10:         return range;
11:     }
12:
13:     public static void main(String[] arguments) {
14:         int[] range;
15:         RangeLister lister = new RangeLister();
16:
17:         range = lister.makeRange(4, 13);
18:         System.out.print("The array: [ ");
19:         for (int i = 0; i < range.length; i++) {
20:             System.out.print(range[i] + " ");
21:         }
22:         System.out.println("]");
23:     }
24:
25: }
```

Run the program by choosing Run, Run File in NetBeans to produce the output shown in Figure 5.1.

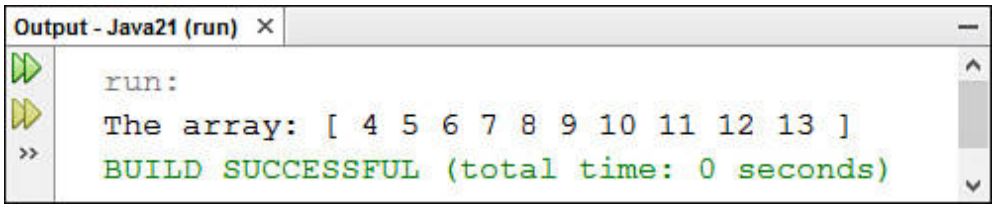


FIGURE 5.1 Using a method to make and display an array.

The `main()` method in this class tests the `makeRange()` method by calling it with the arguments of 4 and 13. The method creates an empty integer array and uses a `for` loop to fill the new array with values from 4 through 13 in lines 7–9.

The `this` Keyword

In the body of a method definition, sometimes you need to refer to the object that contains the method (in other words, the object itself). You can do this to use the object's instance variables and to pass the current object as an argument to another method.

To refer to the object in its own method, use the `this` keyword where you normally would refer to an object's name.

The `this` keyword refers to the current object, and you can use it anywhere a reference to an object might appear: in dot notation, as an argument to a method, as the return value for the current method, and so on. Here are examples of using `this` with comments to explain each one:

Click here to view code image

```
t = this.x;             // the x instance variable for this object

z.resetData(this);      // call the resetData method, defined in
                        // the z class, and pass it the current
object
return this;            // return the current object
```

In many cases, you might not need to explicitly use the `this` keyword because it is assumed. For instance, you can refer to both instance variables and method calls defined in the current class simply by name because the `this` is implicit in those references. Therefore, you could write the first example as follows:

Click here to view code image

```
t = x;                  // the x instance variable for this object
```

Note

The viability of omitting the `this` keyword for instance variables depends on whether variables of the same name are declared in the local scope. You explore this further in the next section.

Because `this` is a reference to the current instance of a class, only use it inside the body of an instance method definition. Class methods—which are declared with the `static` keyword—cannot use `this`.

Variable Scope and Method Definitions

One thing you must know to use a variable is its scope. *Scope* is the part of a program in which a variable exists, making it possible to use the variable in statements and expressions. When the part defining the scope has finished executing, the variable ceases to exist.

When you declare a variable in Java, that variable always has limited scope. A variable with local scope, for example, can be used only inside the block in which it was defined. Instance variables have a scope that extends to the entire class, so they can be used by any of the instance methods within that class.

When you refer to a variable, Java checks for its definition outward, starting with the innermost scope.

The innermost scope could be a block statement, such as the contents of a `while` loop. The second-innermost scope could be the method in which the block is contained.

If the variable hasn't been found in the method, the class itself is checked.

Because of how Java checks for the scope of a given variable, it is possible for you to create a variable in a lower scope that hides (or replaces) the original value of that variable and introduces subtle bugs into your code.

For example, consider the following Java application:

Click here to view code image

```
class ScopeTest {
    int test = 10;

    void printTest() {
```

```
        int test = 20;
        System.out.println("Test: " + test);
    }

    public static void main(String[] arguments) {
        ScopeTest st = new ScopeTest();
        st.printTest();
    }
}
```

This class has two variables with the same name, `test`. The first, an instance variable, is initialized with the value 10. The second is a local variable with the value 20.

The local variable `test` within the `printTest()` method hides the instance variable `test` in that scope. When the `printTest()` method is called within the `main()` method, it displays that `test` equals 20, even though there's a `test` instance variable that equals 10. You could avoid this problem by using `this.test` to refer to the instance variable and using `test` to refer to the local variable.

A more insidious example occurs when you redefine a variable in a subclass that already occurs in a superclass. This can create subtle bugs in your code. For example, you might call methods that are intended to change the value of an instance variable, but the wrong variable is changed. Another bug might occur when you cast an object from one class to another. The value of your instance variable might mysteriously change because the variable was getting that value from the superclass instead of your class.

The best way to avoid this behavior is to be aware of the variables defined in the superclass of your class and avoid duplicating a variable name used higher in the class hierarchy.

Passing Arguments to Methods

When you call a method with an object as a parameter, the object is passed into the method's body as a reference to that object. Any change made to the object inside the method persists outside the method.

Keep in mind that this includes arrays and all objects contained in arrays. When you pass an array into a method and modify its contents, the original array is affected. Primitive types and strings, on the other hand, are passed by value. You can't do anything in the method that changes those types.

The `Passer` class in Listing 5.2 demonstrates how this works. Create this class in NetBeans in the `com.java21days` package.

LISTING 5.2 The Full Text of `Passer.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class Passer {
 4:
 5:     void toUpperCase(String[] text) {
 6:         for (int i = 0; i < text.length; i++) {
 7:             text[i] = text[i].toUpperCase();
 8:         }
 9:     }
10:
11:     public static void main(String[] arguments) {
12:         Passer passer = new Passer();
13:         passer.toUpperCase(arguments);
14:         for (int i = 0; i < arguments.length; i++) {
15:             System.out.print(arguments[i] + " ");
16:         }
17:         System.out.println();
18:     }
19: }
```

This application takes one or more command-line arguments and displays them in all uppercase letters.

In NetBeans, set the arguments by choosing Run, Set Project Configuration, Customize. The Project Properties dialog appears. Enter `com.java21days.Passer` as the Main Class and `Athos Aramis Porthos` (or words of your choosing) as the Arguments and click OK. Run the application by choosing Run, Run Project.

If you use the suggested arguments, the program produces the output shown in Figure 5.2.

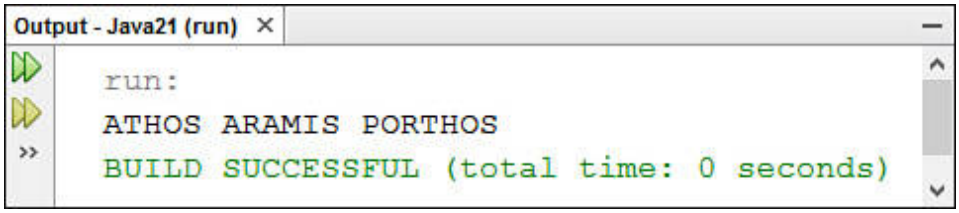


FIGURE 5.2 Testing how objects are passed to a method.

The Passer application uses command-line arguments stored in the `arguments` array of strings.

The application creates a `Passer` object and calls its `toUpperCase()` method with the `arguments` array as an argument (lines 12–13).

Because a reference to the array object is passed to the method, changing the value of each array element in line 7 changes the actual element (rather than a copy of it). Displaying the array with lines 14–16 demonstrates this.

Caution

If nothing happens when you run the Passer application in NetBeans, you're running it with the command Run, Run File instead of Run, Run Project. The Run File command does not use the arguments set up in the project configuration. The Run Project command does.

Class Methods

The relationship between class and instance variables is directly comparable to how class and instance methods work.

Class methods are available to any instance of the class itself and can be made available to other classes. In addition, unlike an instance method, a class does not require an object of the class for its methods to be called.

For example, the Java Class Library includes the `System` class, which defines a set of methods that are useful when displaying text, retrieving configuration information, and accomplishing other tasks. Here are two statements that use its class methods:

Click here to view code image

```
System.exit(0);

long now = System.currentTimeMillis();
```

The `exit(int)` method closes an application with a status code that indicates success (0) or failure (any other value). The `currentTimeMillis()` method returns a `long` holding the number of milliseconds since midnight on Jan. 1, 1970. This number is a representation of the current date and time.

To define class methods, use the `static` keyword in front of the method definition as you would in front of a class variable. For example, the class

method `exit()` in the preceding example might have the following signature:

Click here to view code image

```
static void exit(int argument) {
    // body of method
}
```

Java supplies wrapper classes such as `Integer` and `Float` for each of the primitive types. By using class methods defined in those classes, you can create objects for primitive types, and vice versa. The same value is represented in either form.

For example, the `parseInt()` class method in the `Integer` class can be used with a string argument, returning an `int` representation of that string:

Click here to view code image

```
int count = Integer.parseInt("42");
```

In this statement, `parseInt()` returns the `String` value "42" as an integer with a value of 42, which is stored in the `count` variable.

The lack of a `static` keyword in front of a method name makes it an instance method. Instance methods operate in a particular object, rather than a class of objects. On Day 1, "Getting Started with Java," you created an instance method called `checkTemperature()` that checked the temperature in the robot's environment.

Tip

Methods that affect a particular object should be defined as instance methods. Methods that provide some general capability but do not directly affect an object of the class should be declared as class methods.

Class methods, unlike instance methods, are not inherited. A class method in a superclass cannot be overridden in a subclass.

Creating Java Applications

Now that you know how to create classes, objects, class and instance variables, and class and instance methods, you can put them all together in a Java program.

A Java application consists of one or more classes and can be as large or as small as you want it to be. Although all the applications you've created up to this point do nothing visually other than display characters, you also can create Java applications that use windows, graphics, and a graphical user interface.

The only thing you need to make a Java application run is one class that serves as the starting point.

The class needs only one thing: a `main()` method. When the application is run, the Java Virtual Machine (JVM) calls this method.

The signature for the `main()` method takes the following form:

Click here to view code image

```
public static void main(String[] arguments) {
    // body of method
}
```

Here's a rundown of the parts of the `main()` method:

- `public` means that this method is available to other classes and objects, which is a form of access control. The `main()` method must be declared `public`. You learn more about access methods during Day 6.
- `static` means that `main()` is a class method.
- `void` means that the `main()` method doesn't return a value.
- `main()` takes one parameter, which is an array of strings. This argument holds command-line arguments.

The body of the `main()` method contains any code you need to start your application, such as the initialization of variables or the creation of objects.

The `main()` method is a class method. An object of the class that holds `main()` is not created automatically when your application runs. If you want to treat that class as an object, you have to create an instance of it in the `main()` method (as you did in the Passer application in Listing 5.2 on line 12).

In a NetBeans project, one class can be designated as the main class of the project. When the project is packaged into a single Java archive (JAR) file, the main class will be run if the JAR file is executed.

To set the main class, choose Run, Set Project Configuration, Customize. In the Project Properties dialog, enter the name of this class in the Main Class field.

Helper Classes

Your Java application may consist of a single class—the one with the `main()` method—or several classes that use each other. (In reality, even a simple tutorial program uses numerous classes in the Java Class Library.) You can create as many classes as you want for your program.

As long as Java can find the class, your program uses it when it runs. Note, however, that only the starting-point class needs a `main()` method. After it is called, the methods inside the various classes and objects used in your program take over. Although you can include `main()` methods in helper classes, they are ignored when the program runs.

Java Applications and Arguments

Because Java applications are standalone programs, it's useful to pass arguments to an application to customize how it operates.

You can use arguments to determine how an application will run or to enable an application to operate on different kinds of input. You can use program arguments for many purposes, such as to turn on debugging input or to indicate a filename to load.

Passing Arguments to Java Applications

How you pass arguments to a Java application varies based on the environment and JVM on which Java is being run.

To pass arguments to a Java program with the `java` interpreter included with the Java Development Kit (JDK), the arguments would be appended to the command line when the program is run. For example:

```
java Echo April 450 -10
```

Here `java` is the name of the interpreter, `Echo` is the Java application, and the rest are three arguments passed to a program: "April", "450", and "-10". Note that a space separates each of the arguments.

To group arguments that have spaces in them, surround the arguments with quotation marks. For example, consider the following command line:

Click here to view code image

```
java Echo Wilhelm Niekro Hough "Tim Wakefield" 49
```

Putting quotation marks around "Tim Wakefield" causes that text to be treated as a single argument. The Echo application would receive five arguments: "Wilhelm", "Niekro", "Hough", "Tim Wakefield", and "49". The quotation marks prevent the space within "Tim Wakefield" from being used to separate arguments. Those spaces are not included as part of the argument when it is sent to the program and received using the `main()` method.

Caution

One thing quotation marks are not used for is to identify strings. Every argument passed to an application is stored in an array of `String` objects, even if it has a numeric value (such as 450, –10, and 49 in the preceding examples).

Because NetBeans runs the JVM behind the scenes, there's no command line on which to specify arguments. Instead, they can be set in the project configuration with the Run, Set Project Configuration, Customize command, as you did earlier to run the RangeLister application.

Handling Arguments in Your Java Application

When an application is run with arguments, Java stores the arguments as an array of strings and passes the array to the application's `main()` method. Take another look at the signature for `main()`:

Click here to view code image

```
public static void main(String[] arguments) {
    // body of method
}
```

Here, *`arguments`* is the name of the array of strings that contains the list of arguments. You can call this array anything you want.

Inside the `main()` method, you handle the arguments your program was given by looping through the array. The `Averager` class in Listing 5.3 is a Java application that takes numeric arguments and returns the sum and average of those arguments.

Create a new empty Java file in NetBeans for the `Averager` class in the `com.java21days` package.

LISTING 5.3 The Full Text of `Averager.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class Averager {
 4:     public static void main(String[] arguments) {
 5:         int sum = 0;
 6:
 7:         if (arguments.length > 0) {
 8:             for (int i = 0; i < arguments.length; i++) {
 9:                 sum += Integer.parseInt(arguments[i]);
10:             }
11:             System.out.println("Sum is: " + sum);
12:             System.out.println("Average is: " +
13:                 (float) sum / arguments.length);
14:         }
15:     }
16: }
```

Before running the application in NetBeans, choose two or more numeric arguments in the project configuration, as you did with the RangeLister application. They all should be integers.

The Averager application makes sure that in line 7 at least one argument is passed to the program. This is handled through `length`, the instance variable that contains the number of elements in the `arguments` array.

You must always do things like this when dealing with command-line arguments. Otherwise, your programs crash with `ArrayIndexOutOfBoundsException` errors whenever the user supplies fewer command-line arguments than you were expecting.

If at least one argument is passed to the application, the `for` loop iterates through all the strings stored in the `arguments` array (lines 8–10).

Because all command-line arguments are passed to a Java application as `String` objects, you must convert them to numeric values before using them in any mathematical expressions. The `parseInt()` class method of the `Integer` class takes a `String` object as input and returns an `int` (line 9).

If `75 1080 95 1316` were submitted as your arguments, you would see output matching Figure 5.3.

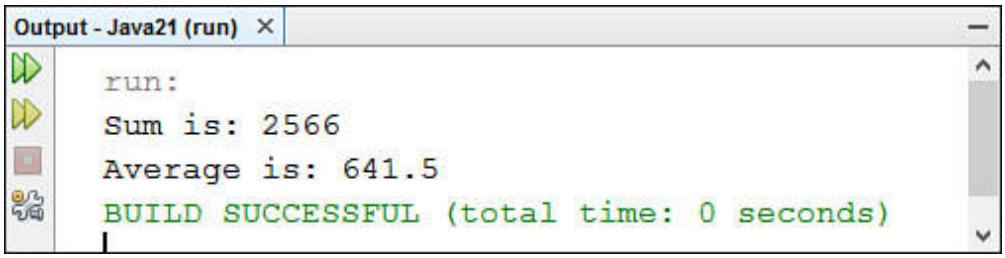


FIGURE 5.3 Receiving arguments in an application.

Creating Methods with the Same Name

When you work with the Java Class Library, you often encounter classes that have numerous methods with the same name.

Two things differentiate these same-named methods:

- The number of arguments they take
- The primitive type or objects of each argument

These two things are part of a method's signature. Using several methods with the same name and different signatures is called *overloading*.

Method overloading can eliminate the need for entirely different methods that do essentially the same thing. Overloading also makes it possible for methods to behave differently based on the arguments they receive.

When you call a method in an object, Java matches the method name and arguments to choose which method definition to execute.

To create an overloaded method, you create different method definitions in a class, each with the same name but different argument lists. The difference can be the number, the type of arguments, or both. Java allows method overloading as long as each argument list is unique for the same method name.

Caution

Java does not consider the return type when differentiating among overloaded methods. If you attempt to create two methods with the same signature and different return types, the class won't compile. In addition, the variable names that you choose for each argument to the method are irrelevant. The number and the type of arguments are the two things that matter.

The next project you undertake creates an overloaded method. It begins with a simple class definition for a class called `Box`. This defines a rectangular shape with four instance variables to define the upper-left and lower-right corners of the rectangle, (`x1`, `y1`) and (`x2`, `y2`):

```
class Box {
    int x1 = 0;
    int y1 = 0;
    int x2 = 0;
    int y2 = 0;
}
```

When a new instance of the `Box` class is created, all its instance variables are initialized to 0.

A `buildBox()` instance method sets the variables to their correct values:

Click here to view code image

```
Box buildBox(int x1, int y1, int x2, int y2) {
    this.x1 = x1;
    this.y1 = y1;
    this.x2 = x2;
    this.y2 = y2;
    return this;
}
```

This method takes four integer arguments and returns a reference to the resulting `Box` object. Because the arguments have the same names as the instance variables, the keyword `this` is used inside the method when referring to the instance variables.

This method can be used to create rectangles, but what if you wanted to define a rectangle's dimensions differently? An alternative would be to use `Point` objects rather than individual coordinates—because `Point` objects contain both an `x` and `y` value as instance variables.

You can overload `buildBox()` by creating a second version of the method with an argument list that takes two `Point` objects:

Click here to view code image

```
Box buildBox(Point topLeft, Point bottomRight) {
    x1 = topLeft.x;
    y1 = topLeft.y;
    x2 = bottomRight.x;
    y2 = bottomRight.y;
    return this;
}
```

For this method to work, the `java.awt.Point` class must be imported so that it can be referred to by the short name `Point`.

Another possible way to define the rectangle is to use a top corner, a height, and a width:

Click here to view code image

```
Box buildBox(Point topLeft, int w, int h) {
    x1 = topLeft.x;
    y1 = topLeft.y;
    x2 = (x1 + w);
    y2 = (y1 + h);
    return this;
}
```

To finish this example, a `printBox()` method is created to display the rectangle's coordinates. A `main()` method turns `Box` into an application and tries out everything on a `Box` object. Listing 5.4 shows the completed class definition. Create this class using NetBeans in package `com.java21days`.

LISTING 5.4 The Full Text of `Box.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class Box {
 6:     int x1 = 0;
 7:     int y1 = 0;
 8:     int x2 = 0;
 9:     int y2 = 0;
10:
11:     Box buildBox(int x1, int y1, int x2, int y2) {
12:         this.x1 = x1;
13:         this.y1 = y1;
14:         this.x2 = x2;
15:         this.y2 = y2;
16:         return this;
17:     }
18:
19:     Box buildBox(Point topLeft, Point bottomRight) {
20:         x1 = topLeft.x;
21:         y1 = topLeft.y;
22:         x2 = bottomRight.x;
23:         y2 = bottomRight.y;
```

```
24:         return this;
25:     }
26:
27:     Box buildBox(Point topLeft, int w, int h) {
28:         x1 = topLeft.x;
29:         y1 = topLeft.y;
30:         x2 = (x1 + w);
31:         y2 = (y1 + h);
32:         return this;
33:     }
34:
35:     void printBox(){
36:         System.out.print("Box: <" + x1 + ", " + y1);
37:         System.out.println(", " + x2 + ", " + y2 + ">");
38:     }
39:
40:     public static void main(String[] arguments) {
41:         Box rect = new Box();
42:
43:         System.out.println("Calling buildBox with "
44:             + "coordinates (25,25) and (50,50):");
45:         rect.buildBox(25, 25, 50, 50);
46:         rect.printBox();
47:
48:         System.out.println("\nCalling buildBox with "
49:             + "points (10,10) and (20,20):");
50:         rect.buildBox(new Point(10, 10), new Point(20, 20));
51:         rect.printBox();
52:
53:         System.out.println("\nCalling buildBox with "
54:             + "point (10,10), width 50 and height 50:");
55:
56:         rect.buildBox(new Point(10, 10), 50, 50);
57:         rect.printBox();
58:     }
59: }
```

Run the application to see the output depicted in Figure 5.4.

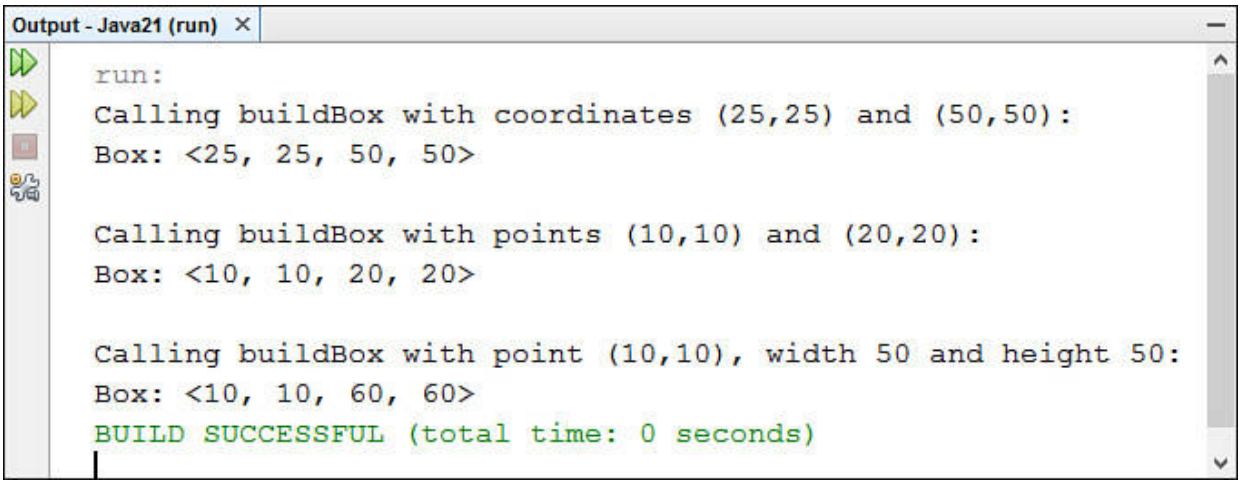


FIGURE 5.4 Calling overloaded methods.

You can define as many versions of a method as you need to implement the behavior needed for that class.

When you have several methods that do similar things, using one method to call another is a shortcut technique to consider. For example, the `buildBox()` method in lines 19–25 can be replaced with the following, much shorter, method:

Click here to view code image

```
Box buildBox(Point topLeft, Point bottomRight) {
    return buildBox(topLeft.x, topLeft.y,
        bottomRight.x, bottomRight.y);
}
```

The `return` statement in this method calls the `buildBox()` method in lines 11–17 with four integer arguments, producing the same result in fewer statements.

This application uses a programming shortcut for working with objects that hasn't been employed up to this point. Take a look at line 56:

Click here to view code image

```
rect.buildBox(new Point(10, 10), 50, 50);
```

The `new` operator is used as an argument to a method. This makes the argument the object created by calling that constructor, which is possible in Java because calling `new` is an expression whose value is the newly created object.

The preceding statement accomplishes the same thing as these two lines of code:

Click here to view code image

```
Point rectangle = new Point(10, 10), 50, 50);
rect.buildBox(new Point(10, 10), 50, 50);
```

The one-line version is more efficient because it doesn't store an object in a variable that will be used only once and never needs to be accessed in any subsequent code. You will find this shortcut employed often in Java programs.

Constructors

You also can define constructors in your class definition that are called automatically when objects of that class are created. A constructor is a method called on an object when it is created—in other words, when it is constructed.

Unlike other methods, a constructor cannot be called directly. Java does three things when `new` is used to create an instance of a class:

- It allocates memory for the object.
- It initializes that object's instance variables, either to initial values or to a default (0 for numbers, `null` for objects, `false` for Booleans, or '\0' for characters).
- It calls a constructor of the class.

If a class doesn't have any constructors defined, an object still is created when the `new` operator is used in conjunction with the class. However, you might have to set its instance variables or call other methods that the object needs to initialize itself.

When an object is created of a class that has no constructors, a constructor with no arguments is implicitly provided by Java. This constructor is called to create the object. For this reason, a constructor with no arguments can be called with `new` even when no constructors are defined.

By defining constructors in your own classes, you can set initial values of instance variables, call methods based on those variables, call methods on other objects, and set an object's initial properties.

When creating a class, you can overload constructors, as you can do with methods, to create an object that has specific properties based on the arguments you give to `new`.

If a class has a constructor that takes one or more arguments, a constructor with no arguments can be called only if one has been defined in the class.

Basic Constructors

Constructors look a lot like regular methods, with three basic differences:

- They always have the same name as the class.
- They don't have a return type.
- They cannot return a value in the method by using the `return` statement.

For example, the following class uses a constructor to initialize its instance variables based on arguments for `new`:

Click here to view code image

```
class MarsRobot {
    String status;
    int speed;
    int power;

    MarsRobot(String in1, int in2, int in3) {
        status = in1;
        speed = in2;
        power = in3;
    }
}
```

You could create an object of this class with the following statement:

Click here to view code image

```
MarsRobot curiosity = new MarsRobot("exploring", 5, 200);
```

The `status` instance variable would be set to "exploring", `speed` to 5, and `power` to 200.

Calling Another Constructor

If you have a constructor that duplicates some of the behavior of an existing constructor, you can call the first constructor from inside the body of the second. Java provides special syntax for doing this. Use the following code to call a constructor defined in the current class:

Click here to view code image

```
this(argument1, argument2, argument3);
```

The use of `this` with a constructor is similar to how `this` can be used to access a current object's variables. In the preceding statement, the arguments with `this()` are the arguments for the constructor.

For example, consider a simple class that defines a circle using the (x, y) coordinate of its center and the length of its radius. The class, `Circle`, could have two constructors: one where the radius is defined and one where the radius is set to a default value of 1. Here's code that does this:

Click here to view code image

```
class Circle {
    int x, y, radius;

    Circle(int xPoint, int yPoint, int radiusLength) {
        this.x = xPoint;
        this.y = yPoint;
        this.radius = radiusLength;
    }

    Circle(int xPoint, int yPoint) {
        this(xPoint, yPoint, 1);
    }
}
```

The second constructor in `Circle` takes only the (x, y) coordinates of the circle's center. Because no radius is defined, the default value of 1 is used. The first constructor is called with the arguments `xPoint`, `yPoint`, and the integer literal 1.

Overloading Constructors

Like methods, constructors also can take varying numbers and types of arguments. This capability enables you to create an object with exactly the properties you want it to have, or as an alternative, allows the object to calculate properties from different kinds of input.

For example, the `buildBox()` methods that you defined in the `Box` class earlier today would make excellent constructors because they are used to initialize an object's instance variables to the appropriate values. So, instead of the original `buildBox()` method you defined (which took four arguments for the corners' coordinates), you could create a constructor.

Listing 5.5 shows a new class, `Box2`, that has the same functionality as the original `Box` class but uses overloaded constructors instead of overloaded

`buildBox()` methods. Create the `Box2` class in NetBeans, putting it in package `com.java21days`.

LISTING 5.5 The Full Text of `Box2.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class Box2 {
 6:     int x1 = 0;
 7:     int y1 = 0;
 8:     int x2 = 0;
 9:     int y2 = 0;
10:
11:     Box2(int x1, int y1, int x2, int y2) {
12:         this.x1 = x1;
13:         this.y1 = y1;
14:         this.x2 = x2;
15:         this.y2 = y2;
16:     }
17:
18:     Box2(Point topLeft, Point bottomRight) {
19:         this(topLeft.x, topLeft.y, bottomRight.x,
20:             bottomRight.y);
21:     }
22:
23:     Box2(Point topLeft, int w, int h) {
24:         this(topLeft.x, topLeft.y, topLeft.x + w,
25:             topLeft.y + h);
26:     }
27:
28:     void printBox() {
29:         System.out.print("Box: <" + x1 + ", " + y1);
30:         System.out.println(", " + x2 + ", " + y2 + ">");
31:     }
32:
33:     public static void main(String[] arguments) {
34:         Box2 rect;
35:
36:         System.out.println("Calling Box2 with coordinates "
37:             + "(25,25) and (50,50):");
38:         rect = new Box2(25, 25, 50, 50);
39:         rect.printBox();
40:
41:         System.out.println("\nCalling Box2 with points "
42:             + "(10,10) and (20,20):");
```

```
43:          rect = new Box2(new Point(10, 10), new Point(20,
20));
44:          rect.printBox();
45:
46:          System.out.println("\nCalling Box2 with 1 point "
47:              + "(10,10), width 50 and height 50:");
48:          rect = new Box2(new Point(10, 10), 50, 50);
49:          rect.printBox();
50:
51:      }
52: }
```

This application produces the same output as the Box application shown in Figure 5.4. In Listing 5.5, the second and third constructors use `this` in lines 19–20 and lines 24–25 to call the first constructor, giving it the task of creating the object with the specified parameters.

Overriding Methods

When you call an object's method, Java looks for that method definition in the object's class. If it doesn't find it, the method is sought in the object's superclass, and on up the class hierarchy until a method definition is found. Inheritance enables you to define and use methods repeatedly in subclasses without having to duplicate the code.

However, there might be times when you want an object to respond to the same method but have different behavior when that method is called. In that case, you can override the method.

To do this, define a method in a subclass with the same signature as a method in a superclass. Then, when the method is called, the subclass method is found and executed instead of the one in the superclass. This is called *overriding* a method.

Creating Methods That Override Existing Methods

To override a method, all you have to do is create a method in your subclass that has the same signature (name and argument list) as a method defined by your class's superclass. Because Java executes the first method definition it finds that matches the signature, the new signature hides the original method definition.

Here's a simple example. Listing 5.6 contains two classes. `Printer` contains a method called `printMe()` that displays information about

objects of that class. `SubPrinter` is a subclass that adds a `z` instance variable to the class. Create this class and name it `Printer` in NetBeans (package `com.java21days`).

LISTING 5.6 The Full Text of `Printer.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class Printer {
 4:     int x = 0;
 5:     int y = 1;
 6:
 7:     void printMe() {
 8:         System.out.println("x is " + x + ", y is " + y);
 9:         System.out.println("I am an instance of the class " +
10:             this.getClass().getName());
11:     }
12: }
13:
14: class SubPrinter extends Printer {
15:     int z = 3;
16:
17:     public static void main(String[] arguments) {
18:         SubPrinter obj = new SubPrinter();
19:         obj.printMe();
20:     }
21: }
```

When this file is compiled, there are two class files rather than one. Because the source file defines the `Printer` and `SubPrinter` classes, the compiler produces both. Run `SubPrinter` (by selecting Run, Run File in NetBeans), and you see the output in Figure 5.5.

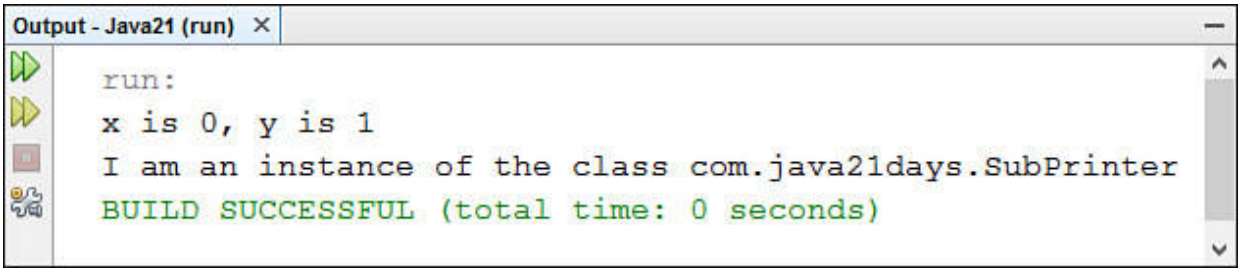


FIGURE 5.5 Calling a superclass method in a subclass.

Caution

The `Printer` class does not have a `main()` method, so it cannot be run as an application. So when you choose Run, Run File in NetBeans, it automatically runs the SubPrinter application's `main()` method, because no other class has such a method. If a source code file contains more than one class with `main()`, NetBeans asks which one should be run.

In the application a `SubPrinter` object was created and the `printMe()` method was called in the `main()` method of `SubPrinter`. Because the `SubPrinter` does not define this method, Java looks for it in the superclasses of `SubPrinter`, starting with `Printer`. `Printer` has a `printMe()` method, so it is executed. Unfortunately, this method does not display the `z` instance variable, as you can see from the preceding output. The superclass does not define this variable, so it could not display it.

To correct the problem, you can override the `printMe()` method in `SubPrinter`, adding a statement to display the `z` instance variable:

Click here to view code image

```
void printMe() {
    System.out.println("x is " + x + ", y is " + y +
        ", z is " + z);
    System.out.println("I am an instance of the class " +
        this.getClass().getName());
}
```

Calling the Original Method

Usually, there are two reasons why you want to override a method that a superclass already has implemented:

- To replace the definition of that original method
- To augment the original method with additional behavior

Overriding a method and giving it a new definition hides the original method definition. However, sometimes behavior should be added to the original definition instead of being replaced, particularly when behavior is duplicated in both the original method and the method that overrides it. By calling the original method in the body of the overriding method, you can add only what you need.

Use the `super` keyword to call the original method from inside a method definition. This keyword passes the method call up the hierarchy, as shown in the following:

Click here to view code image

```
void doMethod(String a, String b) {
    // do stuff here
    super.doMethod(a, b);
    // do more stuff here
}
```

The `super` keyword, similar to the `this` keyword, is a placeholder for the class's superclass. You can use it anywhere that you use `this`, but `super` refers to the superclass rather than to the current object.

Overriding Constructors

Technically, constructors cannot be overridden. Because they always have the same name as the current class, new constructors are created instead of being inherited. This system is fine much of the time; when your class's constructor is called, the constructor with the same signature for all your superclasses also is called. Therefore, initialization can happen for all parts of a class you inherit.

However, when you are defining constructors for your own class, you might want to change how your object is initialized, not only by initializing new variables added by your class, but also by changing the contents of variables that are already there. To do this, explicitly call the constructors of the superclass, and change whatever variables need to be changed.

To call a regular method in a superclass, you use `super.`*`methodname`*`(`*`arguments`*`)`. Because constructor methods don't have a method name to call, the following form is used:

Click here to view code image

```
super(argument1, argument2, ...);
```

Java has a rule for the use of `super()`: It must be the first statement in your constructor definition. If you don't call `super()` explicitly in that first statement, Java automatically calls `super()` with no arguments before the first statement in the constructor.

Because a call to a `super()` method must be the first statement, you can't do something like the following in your overriding constructor:

Click here to view code image

```
if (condition == true) {
    super(1,2,3); // call one superclass constructor
} else {
    super(1,2); // call a different constructor
}
```

Similar to using `this()` in a constructor, `super()` calls the constructor for the immediate superclass (which might, in turn, call the constructor of its superclass, and so on). Note that a constructor with that signature has to exist in the superclass for the call to `super()` to work. The Java compiler checks this when a class is compiled.

You don't have to call the constructor in your superclass that has the same signature as the constructor in your class; you have to call the constructor only for the values you need initialized. In fact, you can create a class that has constructors with entirely different signatures from any of the superclass's constructors.

Listing 5.7 shows a class called `NamedPoint`, which extends the class `Point` from the `java.awt` package. The `Point` class has only one constructor, which takes an `x` and a `y` argument and returns a `Point` object. `NamedPoint` has an additional instance variable (a string for the name) and defines a constructor to initialize `x`, `y`, and the name. Create this class in NetBeans in the `com.java21days` package.

LISTING 5.7 The `NamedPoint` Class

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class NamedPoint extends Point {
 6:     String name;
 7:
 8:     NamedPoint(int x, int y, String name) {
 9:         super(x, y);
10:         this.name = name;
11:     }
```

```
12:
13:     public static void main(String[] arguments) {
14:         NamedPoint np = new NamedPoint(5, 5, "SmallPoint");
15:         System.out.println("x is " + np.x);
16:         System.out.println("y is " + np.y);
17:         System.out.println("Name is " + np.name);
18:     }
19: }
```

The output is displayed in Figure 5.6.

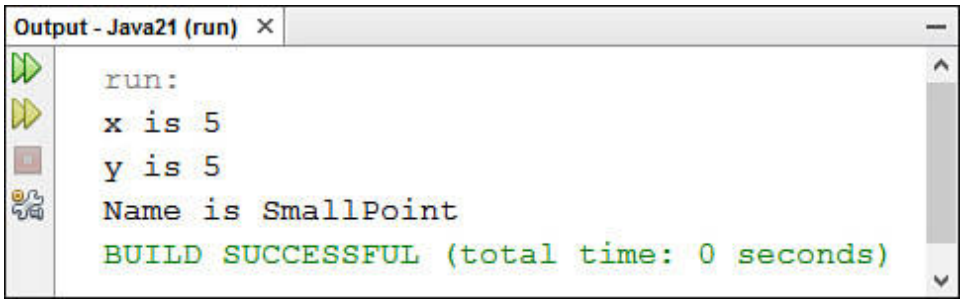


FIGURE 5.6 Extending a superclass constructor in a subclass.

The constructor defined for `NamedPoint` calls `Point`'s constructor to initialize the instance variables of `Point` (`x` and `y`). Although you can just as easily initialize `x` and `y` yourself, you might not know what other things `Point` is doing to initialize itself. Therefore, it is always a good idea to pass constructors up the hierarchy to make sure that everything is set up correctly.

Summary

After finishing today's lesson, you should have a pretty good idea of the relationship among classes in Java and programs you create using the language.

Everything you create in Java involves the use of a main class that interacts with other classes as needed. It's a different programming mindset than you might be used to with other languages.

During this day, you put together everything you have learned about creating Java classes. These topics were covered:

- Instance and class variables, which hold the attributes of a class and objects created from it
- Instance and class methods, which define the behavior of a class. You learned how to define methods, including the parts of a method signature, how to return values from a method, how arguments are

passed to methods, and how to use the `this` keyword to refer to the current object.

- The `main()` method of Java applications, and how to pass arguments to it
- Overloaded methods, which reuse a method name by giving it different arguments
- Constructors, which define the initial variables and other starting conditions of an object

Q&A

Q My class has an instance variable called `origin`. It also has a local variable called `origin` in a method, which, because of variable scope, gets hidden by the local variable. Is there any way to access the instance variable's value?

A The easiest way to avoid this problem is to give your local variables different names than your instance variables. If for some reason you prefer to call a local variable `origin` when there's an instance variable of the same name, you can use `this.origin` to refer to the instance variable and `origin` to refer to the local variable.

Q I created two methods with the following signatures:

Click here to view code image

```
int total(int arg1, int arg2, int arg3) { ... }
float total(int arg1, int arg2, int arg3) { ... }
```

The Java compiler complains when I try to compile the class with these method definitions, even though their signatures are different. What did I do wrong?

A Your methods have the same signature. Method overloading in Java works only if the argument lists are different in either number or type of arguments. Return type is not part of a method signature, so it's not considered when methods have been overloaded. Looking at it from the point at which a method is called, this makes sense: If two methods have exactly the same argument list, how would Java know which one to call?

Q I wrote a program to take four arguments, but when I give it too few arguments, it crashes with a runtime error. Why?

A It's up to you to test for the number and type of arguments your program expects; Java won't do it for you. If your program requires four arguments, test in the `main()` method that you have indeed been given four arguments by using the `length` variable of an array, which contains the count of its elements. Return an error message if you haven't and end the program.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. If a local variable has the same name as an instance variable, how can you refer to the instance variable in the scope of the local variable?

   A. You can't; you should rename one of the variables.

   B. Use the keyword `this` before the instance variable name.

   C. Use the keyword `super` before the name.

2. Where are instance variables declared in a class?

   A. Anywhere in the class

   B. Outside all methods in the class

   C. After the class declaration and above the first method

3. How can you send to a program an argument that includes a space or spaces?

   A. Surround the argument with double quotes.

   B. Separate the arguments with commas.

   C. Separate the arguments with periods.

Answers

1. B. Answer A is a good idea, but variable name conflicts can be a source of subtle errors in your Java programs.
2. B. Customarily, instance variables are declared right after the class declaration and before any methods. It's necessary only that they be

outside all methods.

3. A. The quotation marks are not included in the argument when it is passed to the program.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class BigValue {
    float result;

    public BigValue(int a, int b) {
        result = calculateResult(a, b);
    }

    float calculateResult(int a, int b) {
        return (a * 10) + (b * 2);
    }

    public static void main(String[] arguments) {
        BiggerValue bgr = new BiggerValue(2, 3, 4);
        System.out.println("The result is " + bgr.result);
    }
}

class BiggerValue extends BigValue {

    BiggerValue(int a, int b, int c) {
        super(a, b);
        result = calculateResult(a, b, c);
    }

    // answer goes here
        return (c * 3) * result;
    }
}
```

What statement should replace `// answer goes here` so that the `result` variable equals 312.0?

A. `float calculateResult(int c) {`

B. `float calculateResult(int a, int b) {`

C. `float calculateResult(int a, int b, int c) {`

D. `float calculateResult() {`

The answer is available on the book's website at www.java21days.com. Visit the Day 5 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Modify the `MarsRobot` project from Day 1 so that it includes constructors.
2. Create a class for four-dimensional points called `FourDPoint` that is a subclass of `Point` from the `java.awt` package.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 6. Packages, Interfaces, and Other Class Features

Classes, the templates used to create objects that can store data and accomplish tasks, turn up in everything you do with the Java language.

Today, you extend your knowledge of classes by learning more about how to create them, use them, organize them, and establish rules for how other classes can use them.

The following subjects are covered:

- Controlling access to methods and variables from outside a class
- Finalizing classes, methods, and variables so that their values or definitions cannot be overridden in subclasses
- Creating abstract classes and methods for factoring common behavior into superclasses
- Grouping classes into packages
- Using interfaces to bridge gaps in a class hierarchy

Modifiers

During this week, you have learned how to define classes, methods, and variables in Java. The programming techniques you learn today involve different ways of thinking about how a class is organized. All these techniques use special modifiers in the Java language. *Modifiers* are keywords that you add to those definitions to change their meanings.

The Java language has a wide variety of modifiers:

- Modifiers for controlling access to a class, method, or variable: `public`, `protected`, and `private`
- The `static` modifier for creating class methods and variables
- The `final` modifier for finalizing the implementations of classes, methods, and variables
- The `abstract` modifier for creating abstract classes and methods
- The `synchronized` and `volatile` modifiers, which are used for threads

To add a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples:

Click here to view code image

```
public class RedButton extends javax.swing.JButton {
    // ...
}

private boolean offline;

static final double WEEKS = 9.5;

protected static final int MEANING_OF_LIFE = 42;

public static void main(String[] arguments) {
    // body of method
}
```

If you're using more than one modifier in a statement, you can place them in any order, as long as all modifiers precede the element they are modifying. Be sure to avoid treating a method's return type—such as `void`—as if it were one of the modifiers. The return type must precede the method name, with no modifiers between them.

Modifiers are optional, as you might have recognized after using some of them in the preceding five days. However, there are many good reasons to use them in your programs.

Access Control for Methods and Variables

The modifiers that you will use most often control access to methods and variables: `public`, `private`, and `protected`. These modifiers determine which variables and methods of a class are visible to other classes.

By using access control, you can dictate how your class is used by other classes. Some variables and methods in a class are of use only within the class itself and should be hidden from other classes. This process is called *encapsulation*: An object controls what the outside world can know about it and how the outside world can interact with it. Encapsulation is the process that prevents class variables from being read or modified by other classes. The only way to use these variables is by calling methods of the class if they are available.

The Java language provides four levels of access control: public, private, protected, and a default level specified by using none of these access control modifiers.

Default Access

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version = "0.7a";

boolean processOrder() {
    // ...
    return true;
}
```

A variable or method declared without an access control modifier is available to any other class in the same package. The Java Class Library is organized into packages such as `javax.swing`, a collection of windowing classes for use in graphical user interface programming; and `java.util`, a useful group of utility classes.

Any variable declared without a modifier can be read or changed by any other class in the same package. Any method declared the same way can be called by any other class in the same package. No other classes can access these elements in any way.

This level of access control doesn't control much access, so it's less useful when you begin thinking about how you want a class to be used by other classes.

Private Access

To completely hide a method or variable and keep it from being used by other classes, use the `private` modifier. The only place these methods or variables can be accessed is within their own class.

A private instance variable can be used by methods in its own class but not by objects of any other class. Private methods can be called by other methods in their own class but cannot be called by any others. This restriction also affects inheritance: Neither private variables nor private methods are inherited by subclasses.

Private variables are useful in two circumstances:

- When other classes have no reason to use that variable
- When another class could wreak havoc by changing the variable in an inappropriate way

For example, consider a Java class called `CouponMachine` that generates discounts for an Internet shopping site. A variable in that class called `salesRatio` could control the size of discounts based on product sales. This variable has a big impact on the business's bottom line. If the variable were changed by other classes, `CouponMachine`'s performance would change significantly. To guard against this scenario, you could declare the `salesRatio` variable as private.

The following class uses private access control:

Click here to view code image

```
class Logger {
    private String format;

    public String getFormat() {
        return this.format;
    }

    public void setFormat(String fmt) {
        if ( (fmt.equals("common")) || (fmt.equals("combined")) )
{
            this.format = fmt;
        }
    }
}
```

In this code example, the `format` variable of the `Logger` class is private, so there's no way for other classes to retrieve or set its value directly.

Instead, it's available through two public methods: `getFormat()`, which returns the value of `format`, and `setFormat(String)`, which sets its value.

The latter method contains logic that allows the variable to be set to only "common" or "combined". This demonstrates a benefit of using public methods as the only means of accessing instance variables of a class: The methods can give the class control over how the variable is accessed and limit the values it can take.

Using the `private` modifier is the main way in which an object encapsulates itself. You can't limit the ways in which a class is used without

using `private` to hide variables and methods. Another class is free to change the variables inside a class and call its methods in many possible ways if you don't control access.

A big advantage of privacy is that it lets the implementation of a class change without affecting the users of that class. If you come up with a better way to accomplish something, you can rewrite the class as long as its public methods take the same arguments and return the same kinds of values.

Public Access

In some cases, you might want a method or variable in a class to be completely available to any other class that wants to use it. For example, the `Color` class in the `java.awt` package has public variables for common colors such as `black`. This variable is used when a graphical class wants to use the color black, so `black` should have no access control.

Class variables often are declared to be public. An example is a set of variables in a `Football` class that represent the number of points used in scoring. The `TOUCHDOWN` variable could equal 6, the `FIELD_GOAL` variable could equal 3, and `SAFETY` could equal 2. If these variables are public, other classes could use them in statements such as the following:

Click here to view code image

```
if (yard < 0) {
    System.out.println("Touchdown!");
    score = score + Football.TOUCHDOWN;
}
```

The `public` modifier makes a method or variable completely available to all classes. You have used it in every application you have written so far in their `main()` methods:

Click here to view code image

```
public static void main(String[] arguments) {
    // ...
}
```

The `main()` method of an application has to be public. Otherwise, it could not be called by a Java Virtual Machine (JVM) to run the class.

Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Protected Access

The next level of access control is to limit a method and variable to use by the following two groups:

- Subclasses of a class
- Other classes in the same package

You do so by using the `protected` modifier, as in the following statement:

Click here to view code image

```
protected boolean outOfData = true;
```

Note

You might be wondering how these two groups differ. After all, aren't subclasses part of the same package as their superclass? Not always. An example is the `java.sql.Date` class, which represents calendar dates in a SQL database. It is a subclass of `java.util.Date`, a more generic date class. Protected access differs from default access in this way; protected variables are available to subclasses, even if they aren't in the same package.

This level of access control is useful if you want to make it easier for a subclass to be implemented. Your class might use a method or variable to help the class do its job. Because a subclass inherits much of the same behavior and attributes, it might have the same job to do. Protected access gives the subclass a chance to use the helper method or variable while preventing an unrelated class from trying to use it.

Consider the example of a class called `AudioPlayer` that plays an audio file. `AudioPlayer` has a method called `openSpeaker()`, which interacts with the hardware to prepare the speaker for playing. `openSpeaker()` isn't important to anyone outside the `AudioPlayer` class, so at first glance you might want to make it private. A snippet of `AudioPlayer` might look something like this:

Click here to view code image

```
class AudioPlayer {

    private boolean openSpeaker(Speaker sp) {
        // implementation here
```

```
        }
    }
```

This code works fine if `AudioPlayer` won't be subclassed. But what if later you need a class called `StreamingAudioPlayer` that is a subclass of `AudioPlayer`? That class needs access to the `openSpeaker()` method to override it and provide support for streaming audio devices. You still don't want the method to be generally available to random objects, so it shouldn't be public, but you want any subclasses to have access to it.

Comparing Levels of Access Control

The differences among the various protection types can be confusing, particularly in the case of protected methods and variables. Table 6.1, which summarizes exactly what is allowed where, helps clarify the differences from the least restrictive (public) to the most restrictive (private) forms of protection.

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From any class outside the package	Yes	No	No	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes	No	No

TABLE 6.1 The Different Levels of Access Control

Access Control and Inheritance

One last issue regarding access control for methods involves subclasses. When you create a subclass and override a method, you must consider the access control in place on the original method.

As a general rule, you cannot override a method in Java and make the new method more restrictively controlled than the original. You can, however, make it more public. The method in a subclass can't reduce the visibility of the one it overrides. The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must be either protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.

Methods declared private are not inherited, so the rules don't apply.

Accessor Methods

In many cases, you may have an instance variable in a class that has strict rules for the values it can contain. An example is a `zipCode` variable. A ZIP Code in the United States must be a five-digit number. (There also is a ZIP+4 format that's nine digits.)

To prevent an external class from setting the `zipCode` variable incorrectly, you can declare it private:

```
private int zipCode;
```

However, what if other classes must be able to set the `zipCode` variable for the class to be useful? In that circumstance, you can give other classes access to a private variable by using an accessor method inside the same class as `zipCode`.

An accessor method provides access to a variable that otherwise would be off-limits. By using a method to provide access to a private variable, you can control how that variable is used. In the ZIP Code example, the class could prevent anyone else from setting `zipCode` to an incorrect value.

Often, separate accessor methods to read and write a variable are available. Reading methods have a name beginning with `get`, and writing methods have a name beginning with `set`, as in `setZipCode(int)` and `getZipCode()`.

There's one exception to this convention: If the variable being accessed is a Boolean, the accessor method doesn't begin with `get`. Instead, start it with `is` as in `isValid()` for the `boolean` variable `valid`. Here's an example:

```
private boolean empty;

public boolean isEmpty() {
```

```
        return empty;
    }
```

Using methods to access instance variables is a common technique in object-oriented programming. This approach makes classes more reusable by guarding against improper use.

Static Variables and Methods

A modifier that you already have used in programs is `static`, which was described in detail during Day 5, "Creating Classes and Methods." The `static` modifier is used to create class methods and variables, as in the following example:

Click here to view code image

```
public class Circle {
    public static double PI = 3.14159265F;
    public double radius;

    public double area() {
        return PI * radius * radius;
    }
}
```

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method, as in `Color.black` or `Circle.PI`. You also can use the name of an object belonging to the class, but for class variables and methods, using the class name is better. This approach makes it more clear what kind of variable or method you're working with; instance variables and methods never can be referred to by a class name.

The following statements use class variables and methods:

Click here to view code image

```
double circumference = 2 * Circle.PI * radius;
double randomNumber = Math.random();
```

Tip

For the same reason as instance variables, class variables can benefit from being private and limiting their use to accessor methods only.

The first project you undertake today is a class called `InstanceCounter` that uses class and instance variables to keep track of how many objects of that class have been created. In NetBeans, create an empty Java file named `InstanceCounter` in the `com.java21days` package you've been using for most of this book. Enter the code shown in Listing 6.1 in the source code file, and save it when you're done.

LISTING 6.1 The Full Text of `InstanceCounter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: public class InstanceCounter {
 4:     private static int numInstances = 0;
 5:
 6:     protected static int getCount() {
 7:         return numInstances;
 8:     }
 9:
10:     private static void addInstance() {
11:         numInstances++;
12:     }
13:
14:     InstanceCounter() {
15:         InstanceCounter.addInstance();
16:     }
17:
18:     public static void main(String[] arguments) {
19:         System.out.println("Starting with " +
20:             InstanceCounter.getCount() + " objects");
21:         for (int  i = 0; i < 500; ++i) {
22:             new InstanceCounter();
23:         }
24:         System.out.println("Created " +
25:             InstanceCounter.getCount() + " objects");
26:     }
27: }
```

NetBeans attempts to compile a Java class when it is saved or run. If there are no errors, you can run it to see the output contained in Figure 6.1.

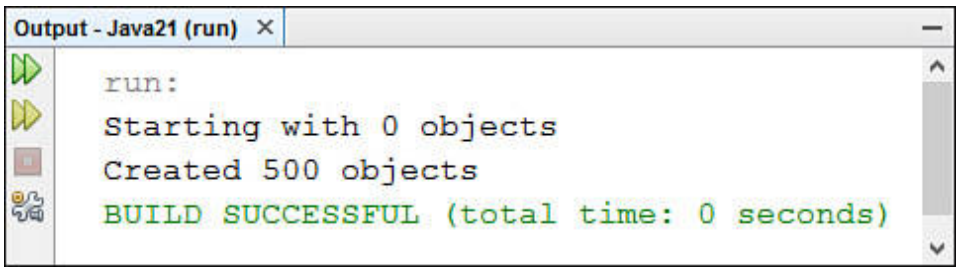


FIGURE 6.1 Working with class and instance variables.

This application demonstrates several features of the Java language. In line 4, a private class variable is declared to hold the number of objects. It is a class variable (declared static) because the number of objects is relevant to the class as a whole, not to any particular object. It's private so that it can be retrieved with only an accessor method.

Note the initialization of `numInstances`. Just as an instance variable is initialized when its instance is created, a class variable is initialized when its class is created. This class initialization happens essentially before anything else can happen to that class, or its instances, so that the class in the example will work as planned.

In lines 6–8, a `get` method is defined so that the private instance variable's value can be retrieved. This method also is declared as a class method because it applies directly to the class variable. The `getCount()` method is declared `protected`, as opposed to `public`, because only this class and perhaps its subclasses are interested in that value; other random classes, therefore, are restricted from seeing it.

Note that there is no accessor method to set the value. The value of the variable should be incremented only when a new instance is created; it should not be set to any random value. Instead of creating an accessor method, a special private method called `addInstance()` is defined in lines 10–12 that increments the value of `numInstances` by 1.

Lines 14–16 create the constructor for this class. Constructors are called when a new object is created, which makes this the most logical place to call `addInstance()` and to increment the variable.

The `main()` method indicates that you can run this as a Java application and test all the other methods. In the `main()` method, 500 objects of the `InstanceCounter` class are created, and then the value of the `numInstances` class variable is displayed.

Final Classes, Methods, and Variables

The `final` modifier is used with classes, methods, and variables to indicate that they will never be changed. It has different meanings for each thing that can be made final, as follows:

- A `final` class cannot be subclassed.
- A `final` method cannot be overridden by any subclasses.
- A `final` variable cannot change in value.

Variables

Final variables often are called constants (or constant variables) because they do not change in value at any time.

With variables, the `final` modifier is used with `static` when making a constant a class variable. If the value never changes, you don't have much reason to give each object in the same class its own copy of the value. They all can use the class variable with the same functionality.

The following statements are examples of declaring constants:

Click here to view code image

```
public static final int TOUCHDOWN = 6;

static final String TITLE = "Captain";
```

Methods

Final methods never can be overridden by a subclass. You declare them using the `final` modifier in the class declaration, as in the following example:

Click here to view code image

```
public final void getSignature() {
    // body of method
}
```

The most common reason to declare a method final is to make the class run more efficiently. Normally, when the JVM runs a method, first it checks the current class to find the method, then it checks its superclass, and so on up the class hierarchy until the method is found. This process sacrifices some speed in the name of flexibility and ease of development.

If a method is final, the Java compiler can put the method's executable bytecode directly into any program that calls the method because the method

will never change because of a subclass that overrides it.

When you first develop a class, you don't have much reason to use `final`. However, if you need to make the class execute more quickly, you can change a few methods into final methods to speed up the process. Doing so removes the possibility that the method later will be overridden in a subclass, so consider this change carefully before continuing.

The Java Class Library declares many of the commonly used methods final so that they can be executed more quickly when used in programs that call them.

Note

Private methods are final without being declared that way because they can't be overridden in a subclass under any circumstance.

Classes

You make a class impossible to subclass by using the `final` modifier in the class's declaration, as in the following:

Click here to view code image

```
public final class ChatServer {
    // body of method
}
```

A `final` class cannot appear after `extends` in a class declaration to create a subclass. As with `final` methods, this process introduces some speed benefits to the Java language at the expense of flexibility.

If you're wondering what you lose by using `final` classes, you must not have tried to subclass anything in the Java Class Library. Many of the popular classes are final, such as `java.lang.String`, `java.lang.Math`, and `java.net.URL`. If you want to create a class that behaves like strings but with some new changes, you can't subclass `String` and define only the behavior that is different. You have to start from scratch.

All methods in a final class are automatically final themselves, so you don't have to use a modifier in their declarations.

Because classes that can provide behavior and attributes to subclasses are much more useful, you should strongly consider whether the benefit of using `final` on one of your classes is outweighed by the cost.

Abstract Classes and Methods

In a class hierarchy, the higher the class, the more abstract its definition. A class at the top of a hierarchy of other classes can define only the behavior and attributes common to all the classes. More-specific behavior and attributes fall somewhere lower down the hierarchy.

When you factor out common behavior and attributes during the process of defining a hierarchy of classes, you might at times find yourself with a class that never needs to be instantiated directly. Instead, such a class serves as a place to hold common behavior and attributes shared by their subclasses.

These classes are called abstract classes, and they are created using the `abstract` modifier. The following is an example:

Click here to view code image

```
public abstract class Palette {
    // ...
}
```

An example of an abstract class is `java.awt.Component`, the superclass of graphical user interface components. Because numerous components inherit from this class, it contains methods and variables useful to each of them. However, there's no such thing as a generic component that can be added to a user interface, so you would never need to create a `Component` object in a program.

Abstract classes can contain anything a normal class can, including constructors, because their subclasses might need to inherit them. Abstract classes also can contain abstract methods, which are method signatures with no implementation. These methods are implemented in subclasses of the abstract class. Abstract methods are declared with the `abstract` modifier. You cannot declare an abstract method in a class that isn't itself abstract. If an abstract class has nothing but abstract methods, you're better off using an interface, as you'll see later today.

Packages

Using packages is a way of organizing groups of classes. A package contains classes that are related in purpose, in scope, or by inheritance.

If your programs are small and use a limited number of classes, you might find that you don't need to explore packages. But as you begin creating more

sophisticated projects with many classes related to each other by inheritance, you might discover the benefit of organizing them into packages.

Packages are useful for several broad reasons:

- They enable you to organize your classes into units. Just as you have folders on your hard drive to organize your files and applications, packages enable you to organize your classes into groups so that you use only what you need for each program.
- They reduce problems with conflicts about names. As the number of Java classes grows, so does the likelihood that you'll use the same class name as another developer. This introduces the possibility of naming clashes and error messages if you try to integrate groups of classes into a single program. Packages provide a way to refer specifically to the desired class, even if it shares a name with a class in another package.
- They enable you to protect classes, variables, and methods in larger ways than on a class-by-class basis. You learn more about protections with packages later.
- Packages can be used to uniquely identify your work.

Every time you use the `import` command or refer to a class by its full package name (`java.util.StringTokenizer`, for example), you are using packages.

To use a class contained in a package, you can use one of three techniques:

- If the class you want to use is in the package `java.lang` (for example, `System` or `Date`), you can use the class name to refer to that class. The `java.lang` classes are automatically available to you in all your programs.
- If the class you want to use is in some other package, you can refer to that class by its full name, including any package names (for example, `java.awt.Font`).
- For classes that you use frequently from other packages, you can import individual classes or a whole package of classes. After a class or package has been imported, you can refer to that class by its class name.

If you don't declare that your class belongs to a package, it is put into an unnamed default package. You can refer to that class and any other unpackaged class by its class name from anywhere in other classes.

To refer to a class in another package, you always can use its full name: the class name preceded by its package. You do not have to import the class or package to use it in this manner:

Click here to view code image

```
java.awt.Font text = new java.awt.Font();
```

For classes that you use only once or twice in your program, using the full name might make sense. If you use a class multiple times, you can import the class to save yourself some typing. But there's no reason not to use import, so you may decide to follow this book's convention of always using it so that class names are short and code is more readable.

The `import` Declaration

To import classes from a package, use the `import` declaration. You can import an individual class, as in this statement:

```
import java.util.ArrayList;
```

You also can import an entire package of classes using an asterisk * in place of an individual class name:

```
import java.awt.*;
```

In an `import` statement, the asterisk can be used only in place of a class name. It does not make it possible to import multiple packages with similar names.

For example, the Java Class Library includes the `java.util`, `java.util.jar`, and `java.util.prefs` packages. You could not import all three packages with the following statement:

```
import java.util.*;
```

This merely imports the `java.util` package. To make all three available in a class, the following statements are required:

```
import java.util.*;
import java.util.jar.*;
import java.util.prefs.*;
```

Also, you cannot indicate partial class names (such as `L*` to import all the classes that begin with the letter L). Your only options when using an

`import` declaration are to identify a single class or use an asterisk to load all the classes in a package.

The `import` declarations in your class definition go at the top of the file, before any class definitions but after the package declaration, as you'll see in the next section.

Using individual `import` declarations or importing packages is mostly a question of your own coding style. Importing a group of classes does not slow down your program or make it any larger; only the classes that you actually use in your code are loaded as they are needed. Importing specific classes makes it easier for readers of your code to figure out what classes are being used in the code.

Note

If you're familiar with C or C++, you might expect the `import` declaration to work like `#include` and potentially result in a large executable program because it includes source code from another file. This isn't the case in Java. The `import` keyword's only function is to tell the Java compiler where to look for the full name of a class when its short name is used. It doesn't actually import code from any classes.

The `import` statement also can be used to refer to constants in a class by name.

Normally, class constants must be prefaced with the name of the class, as in `Color.black`, `Math.PI`, and `File.separator`.

An `import static` statement makes the constants in an identified class available in shorter form. The keywords `import static` are followed by the name of an interface or class and an asterisk. For example:

Click here to view code image

```
import static java.lang.Math.*;
```

This statement makes it possible to refer to the constants in the `Math` class, `E` and `PI`, using only their names. Here's a short example of a class that takes advantage of this feature:

Click here to view code image

```
import static java.lang.Math.*;
```

```
public class ShortConstants {
    public static void main(String[] arguments) {
        System.out.println("PI: " + PI);
        System.out.println("" + (PI * 3));
    }
}
```

Class Name Conflicts

After you have imported a class or a package of classes, you usually can refer to a class by its name without the package identifier. However, you must be more explicit when you import two classes from different packages that have the same class name.

One situation where a naming conflict might occur is during database programming, which you undertake on Day 18, "Accessing Databases with JDBC 4.2 and Derby." This kind of programming can involve the `java.util` and `java.sql` packages, which both contain a class named `Date`.

If you're working with both packages in a class that reads or writes data in a database, you could import them with these statements:

```
import java.sql.*;
import java.util.*;
```

When both these packages are imported, a compiler error occurs when you refer to the `Date` class without specifying a package name, as in this statement:

```
Date now = new Date();
```

The error occurs because the Java compiler has no way of knowing which `Date` class is being referred to in the statement. The package must be included in the statement, like this:

Click here to view code image

```
java.util.Date = new java.util.Date();
```

Tip

NetBeans makes it easy to import individual classes as you write a program. As you enter statements in the source code editor, NetBeans will detect that a class you use hasn't been imported. A warning icon (lightbulb and red circle) appears in the left edge of the editor on that line. When you click the icon, a pop-up menu appears with a command to import the class. Choose it and an `import` statement is added at the top of the class.

Creating Your Own Packages

Creating a package for your classes in Java is not much more complicated than creating a class.

Picking a Package Name

The first step is to decide on a name. The name you choose for your package depends on how you will use those classes. Perhaps you name your package after yourself or a part of the Java system you're working on (such as `graphics` or `messaging`). If you intend to distribute your package as an open source or commercial product, use a package name that uniquely identifies its authorship.

Oracle recommends that Java developers use an Internet domain name that you control as the basis for a unique package name.

To form the name, reverse the elements of the domain so that the last part becomes the first part of the package name, followed by the second-to-last part. Following this convention, because my personal domain name is `cadenhead.org`, Java packages I create begin with the name `org.cadenhead`, such as `org.cadenhead.game` and `org.cadenhead.xml`.

This book's website is at the domain `java21days.com`, so the package for the classes created in its pages is `com.java21days`.

This convention provides a reasonable assurance that no other Java developers will offer a package with the same name, as long as they follow the same rule themselves (as most developers do).

By another convention, package names use no capital letters, which distinguishes them from class names. For example, in the full name of the class `java.lang.String`, you can easily distinguish the package name `java.lang` from the class name `String`.

Creating the Folder Structure

The second step in creating packages is to create a folder structure that matches the package name, which requires a separate folder for each part of the name. The package `org.cadenhead.rss` requires an `org` folder, a `cadenhead` folder inside `org`, and an `rss` folder inside `cadenhead`. The classes in the package then are stored in the `rss` folder.

In NetBeans, when you put a class in a package, the folders are created automatically and the source code and class files are stored in the correct subfolder. All you have to worry about is choosing the package name.

To see this, click the Files tab in the Projects pane to bring it to the front (Figure 6.2). The files and folders of the `Java21` project are shown. You have been using the package `com.java21days` on your programs in this book. Expand the `src` folder, then the `com` subfolder, then the `java21days` subfolder. All the Java source files for classes in this package will be listed.

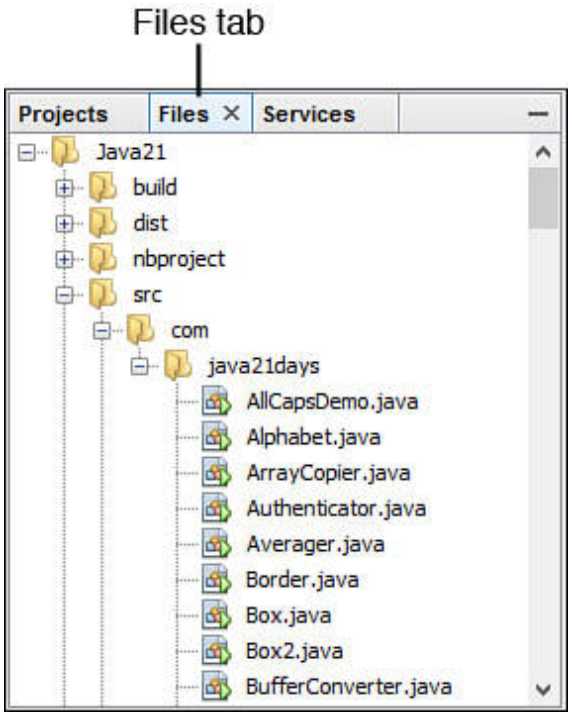


FIGURE 6.2 Viewing a project's package folder hierarchy in NetBeans.

Adding a Class to a Package

The final step of putting a class inside a package is to add a statement above any `import` declarations and the `class` declaration. The `package` declaration is followed by the full name of the package:

```
package org.cadenhead.rss;
```

The `package` declaration must be the first line of code in your source file, disregarding comments or blank lines.

Packages and Class Access Control

Earlier today, you learned about access control modifiers for methods and variables. You also can control access to classes.

Classes have the default access control if no modifier is specified, which means that the class is available to all other classes in the same package but is not visible or available outside that package. The class cannot be imported or referred to by name; classes with package protection are hidden inside the package in which they are contained.

To allow a class to be visible and importable outside your package, you can give it public protection by adding the `public` modifier to its definition:

```
public class Visible {
    // ...
}
```

Classes declared as public can be accessed by other classes outside the package.

Note that when you use an `import` statement with an asterisk, you import only the public classes inside that package. Other classes remain hidden and can be used only by the other classes in that package.

Why would you want to hide a class inside a package? For the same reasons that you want to hide variables and methods inside a class: so that you can have utility classes and behavior that are useful only to your implementation, or so that you can limit your program's interface to minimize the effect of larger changes. As you design your classes, consider the whole package, and decide which classes you want to declare public and which you want to be hidden.

Creating a good package consists of defining a small, clean set of public classes and methods for other classes to use and then implementing them by using any number of hidden support classes. You'll see another use for private classes later today.

Interfaces

Interfaces, like abstract classes and methods, provide templates of behavior that other classes are expected to implement. They also offer significant advantages in class and object design that complement Java's single-inheritance approach to object-oriented programming.

The Problem of Single Inheritance

As you begin turning a project into a hierarchy of classes related by inheritance, you might discover that the simplicity of the class organization is restrictive. This is especially true when you have some behavior that needs to be used by classes that do not share a common superclass.

Other object-oriented programming (OOP) languages include the concept of multiple inheritance, which solves this problem by letting a class inherit from more than one superclass, acquiring behavior and attributes from all its superclasses at once.

This concept makes a programming language more challenging to learn and use. Questions of method invocation and how the class hierarchy is organized become far more complicated with multiple inheritance. They also become more open to confusion and ambiguity.

Because one of the goals of Java was to be simpler than languages that inspired its creation, multiple inheritance was rejected in favor of single inheritance.

A Java interface is a collection of abstract behavior that can be adopted by any class without being inherited from a superclass.

An interface contains abstract method definitions and constants. It has no instance variables or method implementations.

Interfaces are implemented and used throughout the Java Class Library when behavior is expected to be implemented by a number of disparate classes. Later today, you'll use one of the interfaces, `java.lang.Comparable`.

Interfaces and Classes

Classes and interfaces, despite their different definitions, have a great deal in common. Both are declared in source files and compiled into `.class` files. In most cases, an interface can be used anywhere you can use a class.

You can substitute an interface name for a class name in almost every example in this book. Java programmers often say "class" when they actually mean "class or interface." Interfaces complement and extend the power of classes, and the two can be treated almost the same, but an interface cannot be instantiated: `new` can only create an instance of a non-abstract class.

Implementing and Using Interfaces

You can do two things with interfaces: use them in your own classes and define your own. For now, start with using them in your own classes.

To use an interface, include the `implements` keyword as part of your class definition:

Click here to view code image

```
public class AnimatedSign extends Sign
    implements Runnable {
    //...
}
```

In this example, the `Runnable` interface extends the behavior of the `AnimatedSign` class, which is a subclass of `Sign`.

Because interfaces provide nothing but abstract method definitions, you must implement those methods in your own classes using the same method signatures in the interface.

To implement an interface, you must offer all the methods in that interface—you can't pick and choose the methods you need. By implementing an interface, you're telling users of your class that you support the entire interface.

After your class implements an interface, subclasses of your class inherit those new methods and can override or overload them. If your class inherits from a superclass that implements a given interface, you don't have to include the `implements` keyword in your own class definition.

Implementing Multiple Interfaces

Unlike inheritance, where a class can have only one superclass, you can include as many interfaces as you need in a class. Your class must implement the combined behavior of all the included interfaces. To include multiple interfaces in a class, separate their names with commas:

Click here to view code image

```
public class AnimatedSign extends Sign
    implements Runnable, Observer {

    // ...
}
```

Note that complications might arise from implementing multiple interfaces. What happens if two different interfaces both define the same method? You can solve this problem in three ways:

- If the methods in each interface have identical signatures, you implement one method in your class, and that definition satisfies both interfaces.
- If the methods have different argument lists, it is a simple case of method overloading; you implement both method signatures, and each definition satisfies its respective interface definition.
- If the methods have the same argument lists but different return types, you cannot create a method that satisfies both. (Remember that a method signature does not include the method's return type.) In this case, trying to compile a class that implements both interfaces would produce a compiler error message. Encountering this problem suggests that your interfaces have some design flaws that you might need to reexamine.

Other Uses of Interfaces

Almost everywhere that you can use a class, you can use an interface instead. For example, you can declare a variable to be of an interface type:

```
Iterator loop;
```

When a variable is declared to be of an interface type, it must be used to hold an object that implements the interface. Any class that implements the Iterator interface can be stored in `loop`. In this case, because `loop` is an object of

the type `Iterator`, the assumption is that you can call all three of the interface's methods on that object: `hasNext()`, `next()`, and `remove()`.

You can cast objects to an interface, just as you can cast objects to other classes.

Creating and Extending Interfaces

After you use interfaces for a while, the next step is to define your own interfaces. Interfaces look a lot like classes; they are declared in much the same way and can be arranged into a hierarchy. However, you must follow certain rules for declaring them.

New Interfaces

To create a new interface, you declare it like this:

```
interface Expandable {
    // ...
}
```

This declaration is, effectively, the same as a class definition, with the word `interface` replacing the word `class`. Inside the interface definition are methods and variables.

The method definitions inside the interface are public and abstract. You can explicitly declare them as such, or they will be turned into public and abstract methods if you do not include those modifiers. You cannot declare a method inside an interface to be either private or protected.

As an example, here's an `Expandable` interface with one method explicitly declared public and abstract and one declared implicitly:

Click here to view code image

```
public interface Expandable {
    public abstract void expand(); // explicitly public and
abstract
    void contract(); // effectively public and abstract
}
```

Both methods are public and abstract.

Like abstract methods in classes, methods inside interfaces do not have bodies. An interface consists of only method signatures; no implementation is involved.

In addition to methods, interfaces can have variables, but those variables must be declared `public`, `static`, and `final` (making them constant). As with methods, you can explicitly define a variable to be these modifiers or it is implicitly defined as such if you don't use those modifiers. Here's that same `Expandable` definition with two new variables:

Click here to view code image

```
public interface Expandable {
    public static final int INCREMENT = 10;
    long CAPACITY = 15000; // becomes public static and final

    public abstract void expand(); // explicitly public and
abstract
    void contract(); // effectively public and abstract
}
```

Interfaces must have either public or package protection, just like classes. Note, however, that interfaces without the `public` modifier do not automatically convert their methods to public and abstract nor their constants to public. A non-public interface also has non-public methods and constants that can be used only by classes and other interfaces in the same package.

Interfaces, like classes, can belong to a package. Interfaces also can import other interfaces and classes from other packages, just as classes can.

Methods Inside Interfaces

Here's one trick to note about methods inside interfaces: Those methods are supposed to be abstract and apply to any kind of class, but how can you define arguments to those methods? You don't know what class will be using them! The answer lies in the fact that you use an interface name anywhere a class name can be used, as you learned earlier. By defining your method arguments to be interface types, you can create generic arguments that apply to any class that might use this interface.

Consider the interface `Trackable`, which defines methods with no arguments for `track()` and `quitTracking()`. You also might have a method for `beginTracking()`, which has one argument: an object of the `Trackable` class.

What class should that argument be? Any object that implements the `Trackable` interface rather than a particular class and its subclasses. The solution is to declare the argument as simply `Trackable` in the interface:

Click here to view code image

```
public interface Trackable {
    public abstract Trackable beginTracking(Trackable self);
}
```

Then, in an actual implementation for this method in a class, you can take the generic `Trackable` argument and cast it to the appropriate object:

Click here to view code image

```
public class Monitor implements Trackable {

public Trackable beginTracking(Trackable self) {
    Monitor mon = (Monitor) self;
    // ...
    return mon;
}
```

Extending Interfaces

As you can do with classes, you can organize interfaces into a hierarchy. When one interface inherits from another interface, that subinterface acquires all the method definitions and constants that its superinterface declared.

To extend an interface, you use the `extends` keyword just as you do in a class definition:

Click here to view code image

```
interface PreciselyTrackable extends Trackable {
    // ...
}
```

Note that unlike classes, the interface hierarchy has no equivalent of the `Object` class—there is no root superinterface from which all interfaces descend. Interfaces can either exist entirely on their own or inherit from another interface.

Note also that unlike the class hierarchy, the inheritance hierarchy can have multiple inheritance. For example, a single interface can extend as many classes as it needs to (separated by commas in the `extends` part of the definition), and the new interface contains a combination of all its parent's methods and constants.

In interfaces, the rules for managing method name conflicts are the same as for classes that use multiple interfaces; methods that differ only in return type result in a compiler error message.

Creating an Online Storefront

To explore all the topics covered up to this point, the Storefront application uses packages, access control, interfaces, and encapsulation. This application manages the items in an online storefront, handling two main tasks:

- Calculating the sale price of each item, depending on how much of it is presently in stock
- Sorting items according to sale price

The application consists of two classes, `Storefront` and `Item`. These classes will be organized as a new package called `org.cadenhead.ecommerce`.

In NetBeans, choose File, New File, indicate that you're creating a new empty Java file, and then click Next. Give it the class name `Item` and the package name `org.cadenhead.ecommerce`. Click Finish to start entering the code shown in Listing 6.2.

LISTING 6.2 The Full Text of `Item.java`

Click here to view code image

```
 1: package org.cadenhead.ecommerce;
 2:
 3: public class Item implements Comparable {
 4:     private String id;
 5:     private String name;
 6:     private double retail;
 7:     private int quantity;
 8:     private double price;
 9:
10:     Item(String idIn, String nameIn, String retailIn, String
qIn) {
11:         id = idIn;
12:         name = nameIn;
13:         retail = Double.parseDouble(retailIn);
14:         quantity = Integer.parseInt(qIn);
15:
16:         if (quantity > 400)
17:             price = retail * .5D;
18:         else if (quantity > 200)
19:             price = retail * .6D;
20:         else
21:             price = retail * .7D;
22:         price = Math.floor( price * 100 + .5 ) / 100;
23:     }
```

```
24:
25:     public int compareTo(Object obj) {
26:         Item temp = (Item) obj;
27:         if (this.price < temp.price) {
28:             return 1;
29:         } else if (this.price > temp.price) {
30:             return -1;
31:         }
32:         return 0;
33:     }
34:
35:     public String getId() {
36:         return id;
37:     }
38:
39:     public String getName() {
40:         return name;
41:     }
42:
43:     public double getRetail() {
44:         return retail;
45:     }
46:
47:     public int getQuantity() {
48:         return quantity;
49:     }
50:
51:     public double getPrice() {
52:         return price;
53:     }
54: }
```

When you save this file, look in the Projects pane of NetBeans. The `Item.java` source code file has been put in a different place, as shown in Figure 6.3.

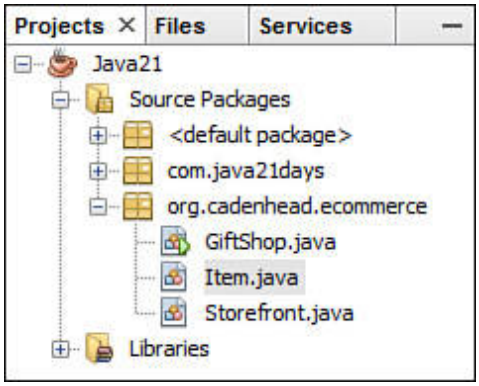


FIGURE 6.3 Grouping packages in a NetBeans project.

NetBeans puts the source code file in the `org.cadenhead.ecommerce` category.

The `Item` class is a support class that represents a product sold by an online store. It contains private instance variables for the product ID code, name, how many are in stock (`quantity`), and the retail and sale prices.

Because all the instance variables of this class are private, no other class can set or retrieve their values. Simple accessor methods are created in lines 35–53 to provide a way for other programs to retrieve these values. Each method begins with `get` followed by the capitalized name of the variable, which is a standard convention in Java programming used throughout the Java Class Library. The `getPrice()` method returns a `double` containing the value of `price`. No methods are provided for setting any of these instance variables. That is handled in the constructor method for this class.

Line 1 establishes that the `Item` class is part of the `org.cadenhead.ecommerce` package.

The `Item` class implements the `Comparable` interface (line 3), which makes it easy to sort a class's objects. This interface has only one method, `compareTo(Object)`, which returns an integer.

The `compareTo()` method compares two objects of a class: the current object and another object passed as an argument to the method. The value returned by the method defines the natural sorting order for objects of this class:

- If the current object should be sorted above the other object, return –1.
- If the current object should be sorted below the other object, return 1.
- If the two objects are equal, return 0.

You determine in the `compareTo()` method which of an object's instance variables to consider when sorting. Lines 25–33 override the `compareTo()` method for the `Item` class, sorting on the basis of the `price` variable. Items are sorted by price from highest to lowest.

After you have implemented the `Comparable` interface for an object, two class methods can be called to sort an array or a class holding those objects. You'll see this later, when the `Storefront` class is created.

The `Item()` constructor in lines 10–23 takes four `String` objects as arguments and uses them to set up the `id`, `name`, `retail`, and `quantity` instance variables. The last two must be converted from strings to numeric

values using the `Double.parseDouble()` and `Integer.parseInt()` class methods, respectively.

The value of the `price` instance variable depends on how much of that item is presently in stock:

- If more than 400 are in stock, `price` is 50 percent of `retail` (lines 16–17).
- If between 201 and 400 are in stock, `price` is 60 percent of `retail` (lines 18–19).
- For everything else, `price` is 70 percent of `retail` (lines 20–21).

Line 22 rounds off `price` so that it contains two or fewer decimal places, turning a value such as $6.92999999999999 into $6.93. The `Math.floor()` method rounds off decimal numbers to the next-lowest integer, returning them as double values.

As the next step in the project, you need a class that represents a storefront for these products. Create an empty Java file with the class name `Storefront` and package name `org.cadenhead.ecommerce`, and enter the code shown in Listing 6.3.

LISTING 6.3 The Full Text of `Storefront.java`

Click here to view code image

```
 1: package org.cadenhead.ecommerce;
 2:
 3: import java.util.*;
 4:
 5: public class Storefront {
 6:     private LinkedList catalog = new LinkedList();
 7:
 8:     public void addItem(String id, String name, String price,
 9:         String quant) {
10:
11:         Item it = new Item(id, name, price, quant);
12:         catalog.add(it);
13:     }
14:
15:     public Item getItem(int i) {
16:         return (Item) catalog.get(i);
17:     }
18:
19:     public int getSize() {
20:         return catalog.size();
```

```
21:     }
22:
23:     public void sort() {
24:         Collections.sort(catalog);
25:     }
26: }
```

Because it belongs to the same package as the `Item` class, Storefront will be listed with it in the NetBeans Projects pane.

The `Storefront` class is used to manage a collection of products in an online store. Each product is an `Item` object, and they are stored together in a `LinkedList` instance variable named `catalog` (line 6).

The `addItem()` method in lines 8–13 creates a new `Item` object based on four arguments sent to the method: the ID, name, price, and quantity of the item that is in stock. After the item is created, it is added to the `catalog` linked list through a call to its `add(Object)` method with the `Item` object as an argument.

The `getItem()` and `getSize()` methods provide an interface to the information stored in the private `catalog` variable. The `getSize()` method in lines 19–21 calls the `catalog.size()` method, which returns the number of objects contained in `catalog`.

Because objects in a linked list are numbered like arrays and other data structures, you can retrieve them using an index number. The `getItem()` method in lines 15–17 calls `catalog.get(int)` with an index number as an argument, returning the object stored at that location in the linked list.

The `sort()` method in lines 23–25 is where you benefit from the implementation of the `Comparable` interface in the `Item` class. The class method `Collections.sort()` sorts a linked list and other data structures based on the natural sort order of the objects they contain, calling the object's `compareTo()` method to determine this order.

To finish this project, the GiftShop application is a class that makes use of `Item` and `Storefront` objects. This application also belongs to the `org.cadenhead.ecommerce` package. Create the new Java class `GiftShop` with the source code shown in Listing 6.4.

LISTING 6.4 The Full Text of `GiftShop.java`

Click here to view code image

```
 1: package org.cadenhead.ecommerce;
 2:
 3: public class GiftShop {
 4:    public static void main(String[] arguments) {
 5:         Storefront store = new Storefront();
 6:         store.addItem("C01", "MUG", "9.99", "150");
 7:         store.addItem("C02", "LG MUG", "12.99", "82");
 8:         store.addItem("C03", "MOUSEPAD", "10.49", "800");
 9:         store.addItem("D01", "T SHIRT", "16.99", "90");
10:         store.sort();
11:
12:         for (int i = 0; i < store.getSize(); i++) {
13:             Item show = (Item) store.getItem(i);
14:             System.out.println("\nItem ID: " + show.getId() +
15:                 "\nName: " + show.getName() +
16:                 "\nRetail Price: $" + show.getRetail() +
17:                 "\nPrice: $" + show.getPrice() +
18:                 "\nQuantity: " + show.getQuantity());
19:         }
20:     }
21: }
```

The `GiftShop` class demonstrates each part of the public interface that the `Storefront` and `Item` classes make available. You can do the following:

- Create an online store
- Add items to it
- Sort the items by sale price
- Loop through a list of items to display information about each one

The output is shown in Figure 6.4.

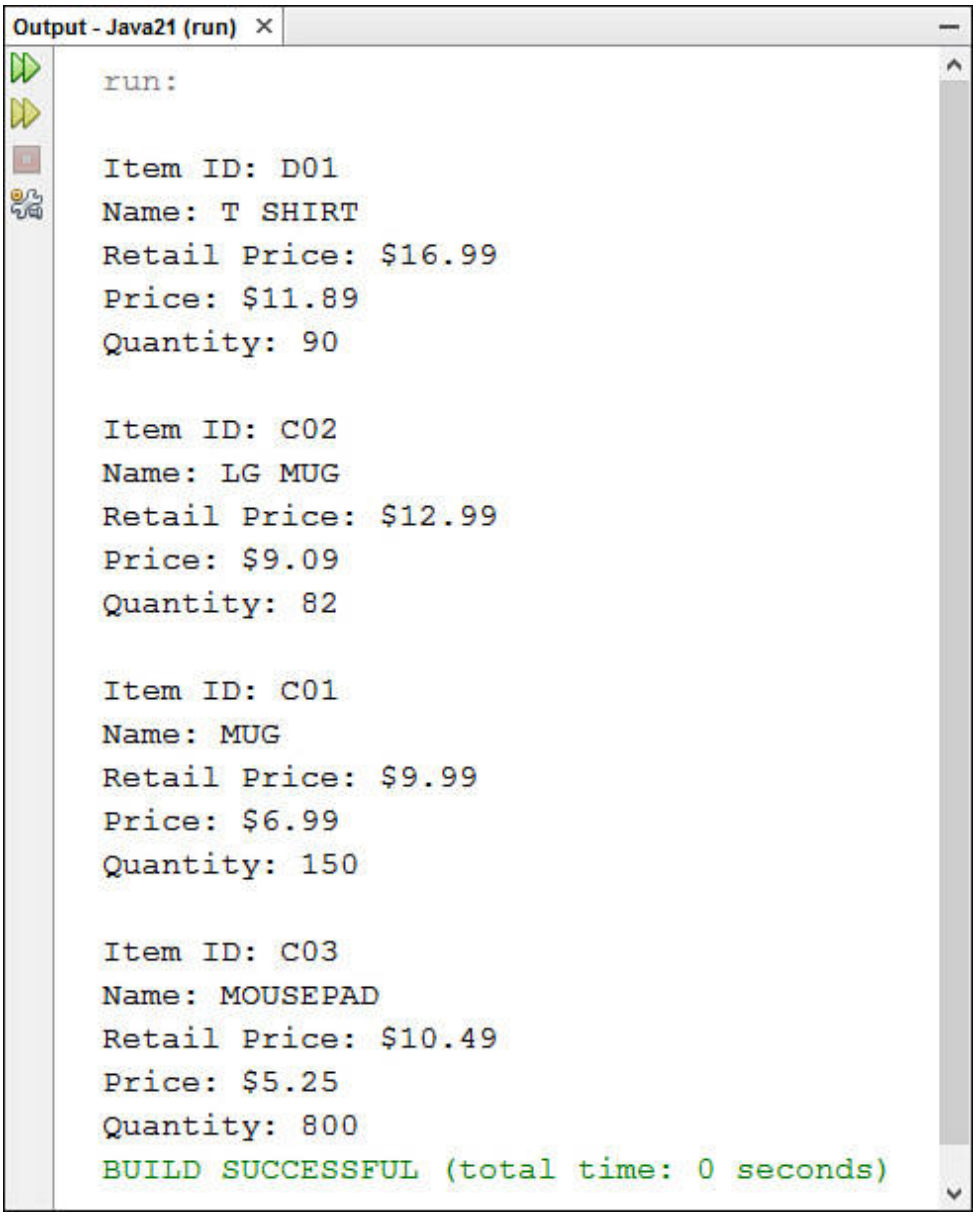


FIGURE 6.4 Displaying a gift shop's inventory sorted by price.

Many implementation details of these classes are hidden from `GiftShop` and other classes that would use the package.

For instance, the programmer who developed `GiftShop` doesn't need to know that `Storefront` uses a linked list to hold all the store's product data. If the developer of `Storefront` later decided to use a different data structure, as long as `getSize()` and `getItem()` returned the expected values, `GiftShop` would continue to work correctly.

Summary

Today you learned how to encapsulate an object by using access control modifiers for its variables and methods. You also learned how to use other modifiers such as `static`, `final`, and `abstract` to develop Java classes and class hierarchies.

To further the effort of developing and using a set of classes, you learned how to group classes into packages. These groupings better organize your programs and help you share classes with the many other Java programmers making their code publicly available.

Finally, you learned how to implement interfaces and inner classes, an extremely helpful Java language feature that models behavior outside of a class hierarchy.

Q&A

Q Won't using accessor methods everywhere slow down my Java code?

A Not always. As Java compilers improve and can implement better optimizations, they will be able to make accessor methods fast automatically. But if you're concerned about speed, you can always declare accessor methods to be `final`, and they'll be comparable in speed to direct instance variable accesses under most circumstances.

Q Based on what I've learned, private abstract methods and final abstract methods and classes don't seem to make sense. Are they legal?

A No. They cause compiler errors, as you have guessed. To be useful, abstract methods must be overridden, and abstract classes must be subclassed, but neither of those operations would be legal if they were also `private` or `final`.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What packages are automatically imported into your Java classes?

A. None

B. The classes stored in the folders of your `Classpath`

C. The classes in the `java.lang` package

2. According to the convention for naming packages, what should be the first part of the name of a package you create?

A. Your name followed by a period

B. Your top-level Internet domain followed by a period

C. The text `java` followed by a period

3. If you create a subclass and override a public method, what access modifiers can you use with that method?

A. `public` only

B. `public` or `protected`

C. `public`, `protected`, or default access

Answers

1. C. All other packages must be imported if you want to use short class names such as `LinkedList` instead of full package and class names such as `java.util.LinkedList`.
2. B. This convention assumes that all Java package developers will own an Internet domain or have access to one so that the package can be made available for download.
3. A. All public methods must remain public in subclasses. Access control in a subclass can be more public or the same as its subclass, but it can't be more private.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
package org.cadenhead.bureau;
```

```
public class Information {
    public int duration = 12;
    protected float rate = 3.15F;
    float average = 0.5F;
}
```

and:

Click here to view code image

```
package org.cadenhead.bureau;

public class MoreInformation extends Information {
    public int quantity = 8;
}
```

and:

Click here to view code image

```
package org.cadenhead.bureau.us;

import org.cadenhead.bureau.*;

public class EvenMoreInformation extends MoreInformation {
    public int quantity = 9;

    EvenMoreInformation() {
        super();
        int i1 = duration;
        float i2 = rate;
        float i3 = average;
    }
}
```

Which instance variables are visible in the `EvenMoreInformation` class?

A. `quantity`, `duration`, `rate`, and `average`

B. `quantity`, `duration`, and `rate`

C. `quantity`, `duration`, and `average`

D. `quantity`, `rate`, and `average`

The answer is available on the book's website at www.java21days.com. Visit the Day 6 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a modified version of the `Storefront` project that includes a `noDiscount` variable for each item. When this variable is `true`, sell the item at the retail price.
2. Create a `ZipCode` class that uses access control to ensure that its `zipCode` instance variable always has a five-digit value.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 7. Exceptions and Threads

Your first week in the Java language ends with two of its most useful elements, threads and exceptions.

Threads are objects that implement the `Runnable` interface or extend the `Thread` class, indicating that they can run simultaneously with other parts of a Java program. Exceptions are objects that represent errors that may occur as a Java program runs.

Threads enable programs to make efficient use of resources by isolating the computing-intensive parts of a program so that they don't slow down everything else. Exceptions enable programs to recognize errors and respond to them. Exceptions even make it possible for programs to correct the conditions and continue running, when possible.

Exceptions are covered first because they're one of the things you use when working with threads.

Exceptions

Programmers in any language endeavor to write programs that are bug-free, never crash, can handle any circumstance with grace, and always recover from unusual situations.

So much for that idea.

Errors occur because programmers didn't anticipate possible problems or didn't test enough. Or programs encounter situations out of their control, such as bad data from users, corrupt files that don't have the correct data in them, network connections that don't connect, hardware devices that don't respond, sunspots, gremlins, and on and on.

In Java, the strange events that might cause a program to fail are called *exceptions*. Java defines a number of language features that deal with exceptions:

- How to handle exceptions in your code and recover gracefully from potential problems
- How to tell code that uses your classes that you're expecting a potential exception

- How to create an exception if you detect one
- How your code is limited yet made more robust by exceptions

With most programming languages, handling error conditions requires much more work than handling a program that is running properly. It can require a confusing structure of conditional statements to deal with errors that might occur.

As an example, consider the following code that could be used to load a file from disk. File input and output can be problematic because of disk errors, file-not-found errors, and the like. If the program must have the data from the file to operate properly, it must deal with all these circumstances before continuing.

Here's the structure of one possible solution:

Click here to view code image

```
int status = loadTextFile();
if (status != 1) {
    // something unusual happened; report it
    switch (status) {
        case 2:
            System.out.println("File not found");
            break;
        case 3:
            System.out.println("Disk error");
            break;
        case 4:
            System.out.println("File corrupted");
            break;
        default:
            System.out.println("Error");
    }
} else {
    // file loaded OK; continue with program
}
```

This code tries to load a file by calling the method `loadTextFile()`, which presumably has been defined elsewhere in the class. The method returns an integer that indicates whether the file loaded properly (a value of 1) or an error occurred (2, 3, 4, or higher).

The program uses a `switch` statement keyed on that error code to address the problem. The end result is a block of code in which the most common circumstance—a successful file load—can be lost amid the error-handling code. This is the result of handling only one possible error. If other errors

take place later in the program, you might end up with more nested `if-else` and `switch-case` blocks.

As you can see, error management would become unmanageable in larger programs, making a Java class difficult to read and maintain.

Dealing with errors in this manner makes it impossible for the compiler to check for consistency the way it can check to make sure that you called a method with the right arguments or set a variable to the right class of object.

Although the previous example uses Java syntax, you never have to deal with errors that way with the Java language. You can use a group of classes called exceptions that work much better.

Exceptions include errors that could be fatal to your program and other circumstances that indicate a problem. By managing exceptions, you can manage errors and possibly work around them.

Errors and other conditions in Java programs can be more easily managed through a combination of language features, consistency checking at compile time, and a set of extensible exception classes.

With these features, you can add a whole new dimension to the behavior and design of your classes, your class hierarchy, and your overall system. Your classes and interface describe how your program is supposed to behave under the best circumstances. With exceptions, you can consistently describe how the program will behave when circumstances are not ideal and allow programmers who use your classes to know what to expect in those cases.

Exception Classes

At this point, it's likely that you've run into at least one Java exception. Maybe you tried to run a Java application without providing the command-line arguments that were needed and saw an `ArrayIndexOutOfBoundsException` message.

Chances are, when an exception occurred, the application quit and spewed a bunch of mysterious errors to the screen. Those errors are exceptions. When your program stops without successfully finishing its work, an exception is thrown. Exceptions can be thrown by the Java Virtual Machine (JVM), by classes you use, or intentionally in your own programs.

Just as exceptions are thrown, they also can be caught. Catching an exception involves dealing with the exceptional circumstance so that your program

doesn't crash, as you learn later today.

Exceptions in Java are instances of classes that inherit from the `Throwable` class. An instance of a `Throwable` class is created when an exception is thrown.

`Throwable` has two subclasses: `Error` and `Exception`. Instances of `Error` are internal errors involving the JVM. These errors are rare and usually fatal to the program; there's not much you can do about them, other than catch them or throw them yourself.

The class `Exception` is more relevant to your own programming. Subclasses of `Exception` fall into two general groups:

- Unchecked exceptions (subclasses of the class `RuntimeException`) such as `ArrayIndexOutofBoundsException`, `SecurityException`, and `NullPointerException`
- Checked exceptions such as `EOFException` and `MalformedURLException`

Unchecked exceptions, also called runtime exceptions, usually occur because of code that isn't very robust. An `ArrayIndexOutofBounds` exception, for example, should never be thrown if you're properly checking to make sure that your code stays within the bounds of an array. `NullPointerException` exceptions happen when you try to use a variable that doesn't refer to an object yet.

Caution

If your program is causing unchecked exceptions, you should fix those problems by improving your code. Don't rely on exception management to handle programming mistakes that can be corrected while you're creating a Java program.

Checked exceptions indicate that something strange and out of control is happening. An `EOFException`, for example, happens when you're reading a file and the file ends before it was expected to. A `MalformedURLException` happens when a web address (also called a URL) isn't in the right format. This group includes exceptions that you create to signal unusual cases that might occur in your own programs.

Exceptions are arranged in a hierarchy just as other classes are, where the superclasses are more general kinds of problems and the subclasses are more specific. This organization becomes more important to you as you deal with exceptions in your own code.

The primary exception classes are part of the `java.lang` package: `Throwable`, `Exception`, and `RuntimeException`. Many of the other packages in the Java Class Library define other exceptions, which are used throughout the library.

The `java.io` package defines a general exception class called `IOException`. It is subclassed not only in the `java.io` package for input and output exceptions (`EOFException` and `FileNotFoundException`) but also in the `java.net` classes for networking exceptions such as `MalformedURLException` and in the `java.util` package with `ZipException`.

Managing Exceptions

Now that you know what an exception is, how do you deal with one in your own code? In many cases, the Java compiler enforces exception management when you try to use methods that throw exceptions; you need to deal with those exceptions in your own code, or it won't compile and NetBeans will flag the error. In this section, you'll learn about consistency checking and how to use three new keywords—`try`, `catch`, and `finally`—to deal with exceptions that might occur.

Exception Consistency Checking

The more you work with the Java Class Library, the more likely you are to run into an exception such as this one:

Output ▼

Click here to view code image

```
Exception java.lang.InterruptedException
must be caught or it must be declared in the throws clause
of this method.
```

In Java, a method can indicate the kinds of errors it might potentially throw. For example, methods that read from files can throw `IOException` errors, so those methods are declared with a special modifier that indicates potential errors. When you use those methods in your own Java programs, you have to protect your code against the exceptions.

This rule is enforced by the compiler itself, in the same way that it checks to make sure that you're using methods with the correct number of arguments and that all your variable types match what you're assigning to them.

Why is this check in place? It makes programs less likely to crash with fatal errors, because you know up front the kind of exceptions that can be thrown by the methods a program uses.

If you define your methods so that they indicate the exceptions they can throw, Java can tell your objects' users to handle those errors.

Protecting Code and Catching Exceptions

Assume that you've been happily coding and an exception occurs as a class is compiled. According to the error message, you have to either catch the error or declare that your method throws it.

First, you deal with catching potential exceptions, which requires two things:

- Protect the code that contains the method that might throw an exception inside a `try` block.
- Handle an exception inside a `catch` block.

A `try` block tries a block of code to see if it can execute all of it without causing an exception. If it fails and an exception occurs, a `catch` block deals with it.

You've seen `try` and `catch` before. On Day 6, "Packages, Interfaces, and Other Class Features," you used the following code to create an integer from a `String` value:

Click here to view code image

```
public SquareTool(String input) {
    try {
        float in = Float.parseFloat(input);
        // rest of method
    } catch (NumberFormatException nfe) {
        System.out.println(input + " is not a valid number.");
```

```
        }
    }
```

In this code, the `Float.parseFloat()` class method might throw an exception of the class `NumberFormatException`, which signifies that the string is not in a valid format as a number. (One situation that triggers this exception is if `input` equals `15x`, which is not a number.)

To handle the exception, the call to `parseFloat()` is placed inside a `try` block, and an associated `catch` block has been set up. The `catch` block receives any `NumberFormatException` objects thrown within the `try` block.

The part of the `catch` clause inside the parentheses is similar to a method definition's argument list. It contains the class of exception to be caught and a variable name. You can use the variable to refer to that exception object inside the `catch` block.

An exception object has a `getMessage()` method that displays a detailed error message describing what happened.

The following example is a revised version of the `try-catch` block used on Day 6:

Click here to view code image

```
try {
    float in = Float.parseFloat(input);
} catch (NumberFormatException nfe) {
    System.out.println("Oops: " + nfe.getMessage());
}
```

The examples you have seen thus far catch a specific type of exception. Because exception classes are organized into a hierarchy and you can use a subclass anywhere that a superclass is expected, you can catch groups of exceptions within the same `catch` statement.

When you write programs that handle input and output from files, Internet servers, and similar places, you deal with several types of `IOException` exceptions (the IO stands for input/output). These exceptions include two of its subclasses, `EOFException` and `FileNotFoundException`. By catching `IOException`, you also catch instances of any `IOException` subclass.

To catch several different exceptions that aren't related by inheritance, you can use multiple `catch` blocks for a single `try`, like this:

Click here to view code image

```
try {
    // code that might generate exceptions
} catch (IOException ioe) {
    System.out.println("Input/output error");
    System.out.println(ioe.getMessage());
} catch (ClassNotFoundException cnfe) {
    System.out.println("Class not found");
    System.out.println(cnfe.getMessage());
} catch (InterruptedException ie) {
    System.out.println("Program interrupted");
    System.out.println(ie.getMessage());
}
```

In a multiple `catch` block, the first `catch` block that matches is executed, and the rest is ignored.

Caution

You can run into unexpected problems by using an `Exception` superclass in a `catch` block followed by one or more of its subclasses in their own `catch` blocks. For example, the input/output exception `IOException` is the superclass of the end-of-file exception `EOFException`. If you put an `IOException` block above an `EOFException` block, the subclass never catches any exceptions.

You also can catch more than one class of exceptions in the same `catch` statement. The classes must be separated by a pipe character `|`. Here's an example:

Click here to view code image

```
try {
    // code that reads a file from disk
} catch (EOFException | FileNotFoundException exc) {
    System.out.println("File error: " + exc.getMessage());
}
```

This code catches two exceptions, `EOFException` and `FileNotFoundException`, in the same `catch` block. The exception is assigned to the `exc` argument, and its `getMessage()` method is called.

The first class in the list that matches the thrown exception will be assigned to the argument.

The exceptions declared as alternatives in the `catch` statement cannot be superclasses or subclasses of each other unless they are in the proper order. The following would not work:

Click here to view code image

```
try {
    // code that reads a file from disk
} catch (IOException | EOFException | FileNotFoundException exc)
{
    System.out.println("File error: " + exc.getMessage());
}
```

This code fails to compile because `IOException` is the superclass of the other two exceptions and it precedes them in the list. Because a superclass can catch exceptions of its subclasses, the second and third exceptions in that statement never would be caught.

Here's a fixed version that would work:

Click here to view code image

```
try {
    // code that reads a file from disk
} catch (EOFException | FileNotFoundException exc) {
    System.out.println("File error: " + exc.getMessage());
} catch (IOException ioe) {
    System.out.println("IO error: " + ioe.getMessage());
}
```

Another way to make it work would be to put the superclass last in the `catch` statement:

Click here to view code image

```
try {
    // code that reads a file from disk
} catch (EOFException | FileNotFoundException | IOException exc)
{
    System.out.println("File error: " + exc.getMessage());
}
```

Caution

Exceptions have a `printStackTrace()` method that displays the sequence of method calls that led to the statement that generated the exception. If you use this in a program, NetBeans flags it for a warning in the source code editor. The reason is that `printStackTrace()` contains debugging information that for security reasons should not be shared with users after a program has been finished.

A `catch` statement must be needed by the corresponding `try` block. The exception class in `catch` has to be one that could be thrown in that block (or a superclass of one that could be thrown). The compiler will fail with an error otherwise.

For example, if you used `catch` for `FileNotFoundException` in a program that did not read any files, the program would not compile.

The `finally` Clause

Suppose that there is some action in your code that you absolutely must do, no matter what happens, regardless of whether an exception is thrown. This is usually to free some external resource after acquiring it, to close a file after opening it, or something similar.

One example is when you are working with databases, as you do during Day 18, "Accessing Databases with JDBC 4.2 and Derby." The database connection and objects you create to access the database are closed in a `finally` block to free those resources because they're no longer needed.

Although you could put that action both inside a `catch` block and outside it, duplicating the same code in two different places should be avoided as much as possible in your programming.

Instead, put that code inside a special optional block of the `try-catch` statement that uses the keyword `finally`:

```
try {
    readTextFile();
} catch (IOException ioe) {
    // deal with IO errors
} finally {
   closeTextFile();
}
```

Today's first project shows how a `finally` statement can be used inside a method.

The HexReader application in Listing 7.1 reads sequences of two-digit hexadecimal numbers and displays their decimal values. There are three sequences to read:

- `000A110D1D260219`
- `78700F1318141E0C`
- `6A197D45B0FFFFFF`

As you learned on Day 2, "The ABCs of Programming," hexadecimal is a base-16 numbering system in which the single-digit numbers range from 00 (decimal 0) to 0F (decimal 15). Double-digit numbers range from 10 (decimal 16) to FF (decimal 255).

Create this class in NetBeans as an empty Java file in the `com.java21days` package and enter the source code of Listing 7.1.

LISTING 7.1 The Full Text of `HexReader.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: class HexReader {
 4:     String[] input = { "000A110D1D260219 ",
 5:         "78700F1318141E0C ",
 6:         "6A197D45B0FFFFFF " };
 7:
 8:     public static void main(String[] arguments) {
 9:         HexReader hex = new HexReader();
10:         for (int i = 0; i < hex.input.length; i++)
11:             hex.readLine(hex.input[i]);
12:     }
13:
14:     void readLine(String code) {
15:         try {
16:             for (int j = 0; j + 1 < code.length(); j += 2) {
17:                 String sub = code.substring(j, j + 2);
18:                 int num = Integer.parseInt(sub, 16);
19:                 if (num == 255) {
20:                     return;
21:                 }
22:                 System.out.print(num + " ");
23:             }
```

```
24:             } finally {
25:                 System.out.println("**");
26:             }
27:             return;
28:         }
29: }
```

The output of this program is shown in Figure 7.1.

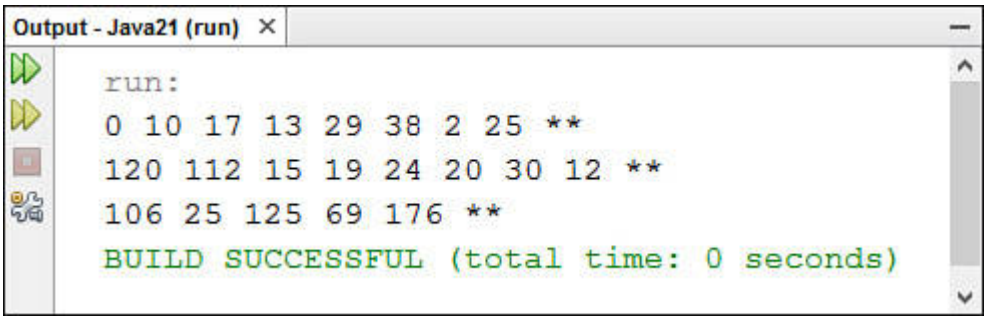


FIGURE 7.1 Displaying decimal values converted from hexadecimal.

Line 17 of the program reads two characters from `code`, the string that was sent to the `readLine()` method, by calling the string's `substring(int, int)` method.

Note

In the `substring()` method of the `String` class, you select a substring in a somewhat counterintuitive way. The first argument specifies the index of the first character to include in the substring, but the second argument does not specify the last character. Instead, the second argument indicates the index of the last character plus 1. A call to `substring(2, 5)` for a string would return the characters from index position 2 to index position 4.

The two-character substring contains a hexadecimal number stored as a `String`. The `Integer` class method `parseInt` can be used with a second argument to convert this number into an integer. Use 16 as the argument for a hexadecimal (base 16) conversion, 8 for an octal (base 8) conversion, and so on.

In the HexReader application, the hexadecimal FF is used to fill out the end of a sequence and should not be displayed as a decimal value. This is accomplished by using a `try-finally` block in lines 15–26 of Listing 7.1.

The `try-finally` block causes an unusual thing to happen when the `return` statement is encountered at line 27. You would expect `return` to cause the `readLine()` method to be exited immediately.

Because it is within a `try-finally` block, the statement within the `finally` block is executed no matter how the `try` block is exited. The text "**" is displayed at the end of a line of decimal values.

There's a way to ensure that resources are freed properly even when an operation inside a `try` block fails with an exception. The `try`-with-resources feature enables statements that claim resources to be declared inside parentheses in a `try` statement.

The following code contains two statements that read data from an Internet server using a networking socket (a type of connection):

Click here to view code image

```
Socket digit = new Socket(host, 79);
BufferedReader in = new BufferedReader(
    new InputStreamReader(digit.getInputStream()));
```

To ensure that resources are properly released, they can be declared inside the `try` statement:

Click here to view code image

```
try (Socket digit = new Socket(host, 79);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(digit.getInputStream()));
    ) {

    // code goes here
} catch (IOException e) {
    System.out.println("IO Error:" + e.getMessage());
}
```

No matter how the code in the `try` block exits, whether through success or an exception, the `digit` and `in` resources will be disposed of properly.

NetBeans issues a warning in the source code editor on any statement that ought to be in a `try`-with-resources statement but isn't. Take this advice whenever you can, because this technique eliminates the common error of forgetting to close a resource when no longer in use.

Declaring Methods That Might Throw Exceptions

You have learned how to deal with methods that might throw exceptions by protecting code and catching any exceptions that occur. The Java compiler checks to make sure that you've dealt with a method's exceptions. But how does it know which exceptions to tell you about?

The answer is that the original method indicated the exceptions that it might possibly throw as part of its definition. You can use this mechanism in your own methods. In fact, it's good style to do so to make sure that users of your classes are alerted to the errors your methods might experience.

To indicate that a method will possibly throw an exception, you use a special clause in the method definition called `throws`.

The `throws` Clause

If some code in your method's body might throw an exception, add the `throws` keyword after the method's closing parenthesis, followed by the name or names of the exception that your method throws. Here's an example:

Click here to view code image

```
public void getPoint(int x, int y) throws NumberFormatException
{
    // body of method
}
```

If your method might throw multiple kinds of exceptions, you can declare them all in the `throws` clause separated by commas:

Click here to view code image

```
public void storePoint(int x, int y)
    throws NumberFormatException, EOFException {
        // body of method
}
```

As with `catch`, you can use a superclass of a group of exceptions to indicate that your method might throw any subclass of that exception. For instance:

Click here to view code image

```
public void loadPoint() throws IOException {
    // body of method
}
```

Keep in mind that adding a `throws` clause to your method definition simply means that the method might throw an exception if something goes wrong, not that it actually will. The `throws` clause provides extra information in your method definition about potential exceptions and allows Java to make sure that your method is being used correctly by other classes.

Think of a method's overall description as a contract between the designer of that method and the caller of the method. (You can be on either side of that contract, of course.)

Usually the description indicates the types of a method's arguments, what it returns, and the particulars of what it normally does. By using `throws`, you are adding information about the abnormal things the method can do. This new part of the contract helps make explicit all the places where exceptional conditions should be handled in your program.

Which Exceptions Should You Throw?

After you decide to declare that your method might throw an exception, you must decide which exceptions it might throw and actually throw them or call a method that will throw them. (You learn about throwing your own exceptions in the next section.)

In many instances, this is apparent from the operation of the method itself. Perhaps you're already creating and throwing your own exceptions, in which case you'll know exactly which exceptions to throw.

You don't have to list all possible exceptions that your method could throw. Unchecked exceptions are handled by the JVM and are so common that you don't have to deal with them.

In particular, exceptions of the `Error` or `RuntimeException` classes or any of their subclasses do not have to be listed in your `throws` clause.

They get special treatment because they can occur anywhere within a Java program and are usually conditions that you, as the programmer, did not directly cause.

One good example is `OutOfMemoryError` when the JVM has run out of memory, which can happen anywhere, at any time, for any number of reasons.

Unchecked exceptions are subclasses of the `RuntimeException` and `Error` classes and are usually thrown by the JVM. You don't have to

declare that your method throws them and usually do not need to deal with them in any other way.

Note

You can choose to list these errors and runtime exceptions in your `throws` clause if you want, but classes that call the method will not be forced to handle them. Only non-runtime exceptions must be handled.

All other exceptions are called *checked exceptions* and are potential candidates for a `throws` clause in your method.

Passing on Exceptions

There are times when it doesn't make sense for your method to deal with an exception. It might be better for the method that calls your method to deal with that exception.

For example, consider the hypothetical example of `WebRetriever`, a class that loads a web page using its web address and stores it in a file. As you learn on Day 17, "Communicating Across the Internet," you can't work with web addresses without dealing with `MalformedURLException`, the exception thrown when an address is in the wrong format.

To use `WebRetriever`, another class calls its constructor with the address as an argument. If the address specified by the other class is in the wrong format, a `MalformedURLException` is thrown. Instead of dealing with this, the constructor of the `WebRetriever` class could have the following declaration:

Click here to view code image

```
public WebRetriever() throws MalformedURLException {
    // body of constructor
}
```

This would force any class that works with `WebRetriever` objects to deal with `MalformedURLException` errors or pass the buck with its own `throws` clause.

One thing is always true: It's better to pass on exceptions to calling methods than to catch them and do nothing in response.

In addition to declaring methods that throw exceptions, there's one other instance in which your method definition may include a `throws` clause: Within that method, you want to call a method that throws an exception, but you don't want to catch or deal with that exception.

Rather than using the `try` and `catch` clauses in your method's body, you can declare your method with a `throws` clause so that it, too, might possibly throw the appropriate exception. It's then the responsibility of the method that calls your method to deal with that exception. This is the other case that tells the Java compiler that you have done something with a given exception.

Using this technique, you could create a method that deals with a `NumberFormatException` without a `try-catch` block:

Click here to view code image

```
public void readFloat(String input) throws NumberFormatException
{
    float in = Float.parseFloat(input);
}
```

After you declare your method to throw an exception, you can use other methods that also throw those exceptions inside the body of this method, without needing to catch the exception.

Note

You can, of course, deal with other exceptions using `try` and `catch` in the body of your method in addition to passing on the exceptions you listed in the `throws` clause. You also can both deal with the exception in some way and then rethrow it so that your method's calling method has to deal with it anyhow.

`throws` and Inheritance

If your method definition overrides a method in a superclass that includes a `throws` clause, there are special rules for how your overridden method deals with `throws`. Unlike other parts of the method signature that must mimic those of the method it is overriding, your new method does not require the same set of exceptions listed in the `throws` clause.

Because there's a possibility that your new method might deal with an exception instead of throwing it, your method can potentially throw fewer types of exceptions. It could even throw no exceptions. This means that you can have the following two class definitions and things will work fine:

Click here to view code image

```
public class RadioPlayer {
    public void startPlaying() throws SoundException {
        // body of method
    }
}
public class StereoPlayer extends RadioPlayer {
    public void startPlaying() {
        // body of method
    }
}
```

The converse of this rule is not true: A subclass method cannot throw more checked exceptions (either exceptions of different types or more general exception classes) than its superclass method.

Any exception thrown by the subclass must be the same as the superclass or a subclass of that exception. Consider this example:

Click here to view code image

```
void readFields() throws IOException {
    // body of method
}
```

If this method is in a superclass and you override the method, this would not be allowed in the subclass:

Click here to view code image

```
void readFiles() throws SQLException {
    // body of method
}
```

`SQLException` is not a subclass of `IOException`, so this code will not compile. But the method could throw `FileNotFoundException`, because that's a subclass of `IOException`.

Creating and Throwing Exceptions

There are two sides to every exception: the side that throws the exception and the side that catches it. An exception can be tossed around a number of times to a number of methods before it's caught, but eventually it will be caught and dealt with.

Many exceptions are thrown by the Java runtime or by methods inside the Java classes themselves. You also can throw any of the standard exceptions that the Java Class Library defines, or you can create and throw your own exceptions.

Throwing Exceptions

Declaring that your method throws an exception is useful to classes that use your method and to the Java compiler, which checks to make sure that all your exceptions are being handled. The declaration itself doesn't do anything to throw that exception should it occur; you must do that as needed in the body of the method.

You need to create a new object of an exception class to throw an exception. After you have that object, use the `throw` statement to throw it.

Here's an example using a hypothetical `NotInServiceException` class that is a subclass of the `Exception` class:

Click here to view code image

```
NotInServiceException nise = new NotInServiceException();
throw nise;
```

You only can throw objects that are subclasses of the `Throwable` class.

Depending on the exception class, the exception also may have arguments to its constructor that you can use. The most common of these is a string argument, which enables you to describe the problem in greater detail (which can be useful for debugging purposes). Here's an example:

Click here to view code image

```
NotInServiceException nise = new
    NotInServiceException("Database Not in Service");
throw nise;
```

After an exception is thrown, the method exits without executing any other code, other than the code inside a `finally` block if one exists. The method

won't return a value either. If the calling method does not have a `try` or `catch` surrounding the call to your method, the program might exit based on the exception you threw.

Creating Your Own Exceptions

Creating new exceptions is easy. Your new exception should inherit from another exception in the Java class hierarchy. All user-created exceptions should be part of the `Exception` hierarchy rather than the `Error` hierarchy, which is reserved for errors involving the JVM. Look for an exception that's close to the one you're creating; for example, an exception for a bad file format would logically be an `IOException`. If you can't find a closely related exception for your new exception, consider inheriting from `Exception`, which sits atop the exception hierarchy for checked exceptions. Unchecked exceptions should inherit from `RuntimeException`.

Exception classes typically have two constructors: The first takes no arguments, and the second takes a single string as an argument.

Exception classes are like other classes. Here's an extremely simple one:

Click here to view code image

```
public class SunSpotException extends Exception {
    public SunSpotException() {}

    public SunSpotException(String message) {
        super(message);
    }
}
```

Combining `throws`, `try`, and `throw`

What if you want to combine the approaches described thus far: You want to handle incoming exceptions in your method, but you also want the option to pass on the exception to your method's caller. Simply using `try` and `catch` doesn't pass on the exception, and adding a `throws` clause doesn't give you a chance to deal with the exception.

If you want to both manage the exception and pass it on to the caller, use all three mechanisms: the `throws` clause, the `try` statement, and a `throw` statement to explicitly rethrow the exception.

Here's a method that uses this technique:

Click here to view code image

```
public void readMessage() throws IOException {
    MessageReader mr = new MessageReader();

    try {
        mr.loadHeader();
    } catch (IOException e) {
        // do something to handle the
        // IO exception and then rethrow
        // the exception ...
        throw e;
    }
}
```

This works because exception handlers can be nested. You handle the exception by doing something responsible with it but decide that it is important enough to give the method's caller a chance to handle it as well.

Exceptions can float all the way up the chain of method callers this way (not being handled by most of them), until finally the JVM handles any uncaught exceptions by aborting your program and printing an error message.

If it's possible for you to catch an exception and do something necessary with it, you should.

When you use `throw` in a `catch` block for an exception superclass, it throws that superclass. This represents a potential loss of information, because the exception could be a subclass with more information about the error.

Here's a situation where that occurs:

- A `try-catch` statement in a file reader looks for an `IOException`.
- An `EOFException` occurs because the end of the file is reached.
- The exception is caught in the `catch` block, because `IOException` is the superclass of `EOFException`.

If `throw` is used with this exception, it will throw an `IOException`, not an `EOFException`. Java 8 introduces a technique that enables the more precise exception to be thrown: Use the `final` keyword in the `catch` statement for the object. This code rewrites the previous example to do this:

```
try {
    mr.loadHeader();
catch (final IOException e) {
```

```
        throw e;
    }
```

Caution

New features in Java 8 require that NetBeans is set up to recognize them or the IDE will flag them as an error. If you enter this code in NetBeans and it displays an error message, make sure your project has been set to the current version of the language. Choose File, Project Properties to open the Project Properties dialog, choose the category `Libraries`, and make sure the Java Platform drop-down is set to `JDK 1.8`.

When Not to Use Exceptions

There are several situations where you should not use exceptions.

First, don't use them in circumstances you could avoid easily in your code. For example, although you can rely on an `ArrayIndexOutofBounds` exception to indicate when you've gone past the end of an array, it's simple to use the array's `length` variable to keep from going beyond the bounds.

In addition, if your users will enter data that must be an integer, testing to make sure that the data is an integer is a much better idea than throwing an exception and dealing with it somewhere else.

Exceptions take up a lot of processing time. A simple conditional will run much faster than exception handling and make your program more efficient. Exceptions should be used only for truly exceptional cases that are out of your control.

It's also easy to get carried away with exceptions and to try to make sure that all your methods have been declared to throw all the possible exceptions that they can throw.

You create more work for everyone involved when you get carried away with exceptions. Declaring a method to throw either few or many exceptions is a trade-off; the more exceptions your method can throw, the more complex that method is to use. Declare only the exceptions that have a reasonably fair chance of happening and that make sense for the overall design of your classes.

Bad Style Using Exceptions

When you first start using exceptions, it might be appealing to work around the compiler errors that result when you use a method that declares a `throws` statement. Although it is permissible to add an empty `catch` clause or to add a `throws` statement to your own method (and there are appropriate reasons for doing so), intentionally dropping exceptions without dealing with them subverts the checks that the Java compiler does for you.

Compiler errors regarding exceptions are there to remind you to reflect on these issues. Take the time to deal with the exceptions that might affect your code. This extra care richly rewards you as you reuse your classes in later projects and in larger and larger programs. The Java Class Library has been written with exactly this degree of care, and that's one of the reasons it's robust enough to be used in your Java projects.

Threads

One thing to consider in Java programming is how system resources are being used. Graphics, complex math computations, and other intensive tasks can take up a lot of processor time.

This is especially true of programs that have a graphical user interface, which is a style of software that you explore next week.

If you write a graphical Java program that does something that consumes a lot of the computer's time, you might find that the program's user interface responds slowly. Drop-down lists take a second or more to appear, button clicks are recognized slowly, and so on.

To solve this problem, you can segregate the processor-hogging functions in a Java class so that they run separately from the rest of the program.

This is possible through the use of threads.

Threads are parts of a program that run on their own while the rest of the program does something else. This also is called *multitasking* because the program handles more than one task simultaneously.

Threads are ideal for anything that takes up a lot of processing time and runs continuously.

By putting the program's hardest workload into a thread, you free up the rest of the program to handle other things. You also make the program easier for the JVM because the most intensive work is isolated.

Writing a Threaded Program

Threads are implemented in Java with the `Thread` class in the `java.lang` package.

The simplest use of threads is to make a program pause in execution and stay idle during that time. To do this, call the `Thread` class method `sleep(long)` with the number of milliseconds to pause as the only argument.

This method throws an exception, `InterruptedException`, when the paused thread has been interrupted for some reason. (One possible reason is when a user closes the program while it is sleeping.)

The following statements stop a program in its tracks for 3 seconds:

Click here to view code image

```
try {
    Thread.sleep(3000);
} catch (InterruptedException ie) {
    // do nothing
}
```

The `catch` block does nothing, which is typical when you're using `sleep()`.

One way to use threads is to put all the time-consuming behavior in its own class.

A thread can be created in two ways: by subclassing the `Thread` class or implementing the `Runnable` interface in another class. Both belong to the `java.lang` package.

Because the `Thread` class implements `Runnable`, both techniques result in objects that start and stop threads in the same manner.

To implement the `Runnable` interface, add the keyword `implements` to the class declaration followed by the name of the interface, as in this example:

Click here to view code image

```
public class StockTicker implements Runnable {
    public void run() {
       // ...
    }
}
```

The `Runnable` interface contains only one method to implement, `run()`.

The first step in creating a thread is to create a reference to an object of the `Thread` class:

```
Thread runner;
```

This statement creates a reference to a thread, but no `Thread` object has been assigned to it yet. Threads are created by calling the constructor `Thread(Object)` with the threaded object as an argument. You could create a threaded `StockTicker` object with the following statement:

Click here to view code image

```
StockTicker tix = new StockTicker();
Thread tickerThread = new Thread(tix);
```

Two good places to create threads are the constructor for an application and the constructor for a component (such as a panel).

A thread is begun by calling its `start()` method, as in the following statement:

```
tickerThread.start();
```

The following statements can be used in a thread class to start the thread:

```
Thread runner = null;
if (runner == null) {
    runner = new Thread(this);
    runner.start();
}
```

The `this` keyword used in the `Thread()` constructor refers to the object in which these statements are contained. The `runner` variable has a value of `null` before any object is assigned to it, so the `if` statement is used to make sure that the thread is not started more than once.

To run a thread, its `start()` method is called. Calling a thread's `start()` method causes another method to be called—the `run()` method that must be present in all threaded objects.

The `run()` method is the engine of a threaded class, containing the processor-intensive behavior and calling methods to perform it.

A Threaded Application

Threaded programming should become more clear when you see it in action.

Listing 7.2 contains `PrimeFinder`, a class that finds a specific prime number in a sequence, such as the 100th, 1,000th, or 30,000th prime. This can take some time, especially for numbers beyond 100,000, so the search for the right prime takes place in its own thread.

Enter the code shown in Listing 7.2 in NetBeans and save it as the class name `PrimeFinder` in the package `com.java21days`.

LISTING 7.2 The Full Text of `PrimeFinder.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: public class PrimeFinder implements Runnable {
 4:     public long target;
 5:     public long prime;
 6:     public boolean finished = false;
 7:     private Thread runner;
 8:
 9:     PrimeFinder(long inTarget) {
10:         target = inTarget;
11:         if (runner == null) {
12:             runner = new Thread(this);
13:             runner.start();
14:         }
15:     }
16:
17:     public void run() {
18:         long numPrimes = 0;
19:         long candidate = 2;
20:         while (numPrimes < target) {
21:             if (isPrime(candidate)) {
22:                 numPrimes++;
23:                 prime = candidate;
24:             }
25:             candidate++;
26:         }
27:         finished = true;
28:     }
29:
30:     boolean isPrime(long checkNumber) {
31:         double root = Math.sqrt(checkNumber);
32:         for (int i = 2; i <= root; i++) {
```

```
33:                 if (checkNumber % i == 0)
34:                     return false;
35:             }
36:             return true;
37:         }
38: }
```

Save the `PrimeFinder` class when you're finished. This class doesn't have a `main()` method, so you can't run it as an application. Next you'll create a program that uses this class.

The `PrimeFinder` class implements the `Runnable` interface, so it can be run as a thread.

There are three public instance variables:

- `target` is a `long` that indicates when the specified prime in the sequence has been found. If you're looking for the 5,000th prime, `target` equals 5000.
- `prime` is a `long` that holds the last prime number found by this class.
- `finished` is a Boolean that indicates when the target has been reached.

There's also a private instance variable called `runner` that holds the `Thread` object this class runs in. This object equals `null` before the thread is started.

The `PrimeFinder` constructor method in lines 9–15 sets the `target` instance variable and starts the thread if it hasn't been started. When the thread's `start()` method is called, it in turn calls the `run()` method of the threaded class.

The `run()` method is in lines 17–28. This method does most of the work of the thread.

This method uses two new variables: `numPrimes`, the number of primes that have been found, and `candidate`, the number that might possibly be prime. The `candidate` variable begins at the first possible prime number, which is 2.

The `while` loop in lines 20–26 continues until the right number of primes has been found.

First, it checks whether the current `candidate` is prime by calling the `isPrime(long)` method, which returns `true` if the number is prime and

`false` otherwise.

If the `candidate` is prime, `numPrimes` increases by 1, and the `prime` instance variable is set to this prime number.

The `candidate` variable is then incremented by 1, and the loop continues.

After the right number of primes has been found, the `while` loop ends, and the `finished` instance variable is set to `true`. This indicates that the `PrimeFinder` object has found the right prime number and is finished searching.

The end of the `run()` method is reached in line 28, and the thread no longer does any work.

The `isPrime()` method is contained in lines 30–37. This method determines whether a number is prime by using the `%` operator, which returns the remainder of a division operation. If a number is evenly divisible by 2 or any higher number (leaving a remainder of 0), it is not a prime number.

Listing 7.3 is an application that uses the `PrimeFinder` class. Enter the code shown in Listing 7.3 in NetBeans as a new Java class named `PrimeThreads` in the `com.java21days` package.

LISTING 7.3 The Full Text of `PrimeThreads.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: public class PrimeThreads {
 4:     public static void main(String[] arguments) {
 5:         PrimeThreads pt = new PrimeThreads(arguments);
 6:     }
 7:
 8:     public PrimeThreads(String[] arguments) {
 9:         PrimeFinder[] finder = new
PrimeFinder[arguments.length];
10:         for (int i = 0; i < arguments.length; i++) {
11:             try {
12:                 long count = Long.parseLong(arguments[i]);
13:                 finder[i] = new PrimeFinder(count);
14:                 System.out.println("Looking for prime " +
count);
15:             } catch (NumberFormatException nfe) {
16:                 System.out.println("Error: " +
nfe.getMessage());
```

```
17:             }
18:         }
19:         boolean complete = false;
20:         while (!complete) {
21:             complete = true;
22:             for (int j = 0; j < finder.length; j++) {
23:                 if (finder[j] == null) continue;
24:                 if (!finder[j].finished) {
25:                     complete = false;
26:                 } else {
27:                     displayResult(finder[j]);
28:                     finder[j] = null;
29:                 }
30:             }
31:             try {
32:                 Thread.sleep(1000);
33:             } catch (InterruptedException ie) {
34:                 // do nothing
35:             }
36:         }
37:     }
38:
39:     private void displayResult(PrimeFinder finder) {
40:         System.out.println("Prime " + finder.target
41:             + " is " + finder.prime);
42:     }
43: }
```

Specify the prime numbers that you're looking for as command-line arguments (using Run, Set Project Configuration, Customize), and include as many as you want.

If this program is run with the command-line arguments `5000 12000 50000 120000`, it is likely to produce the output in Figure 7.2. Because there's no guarantee of the order threads will finish, the report of each found prime may be ordered differently.

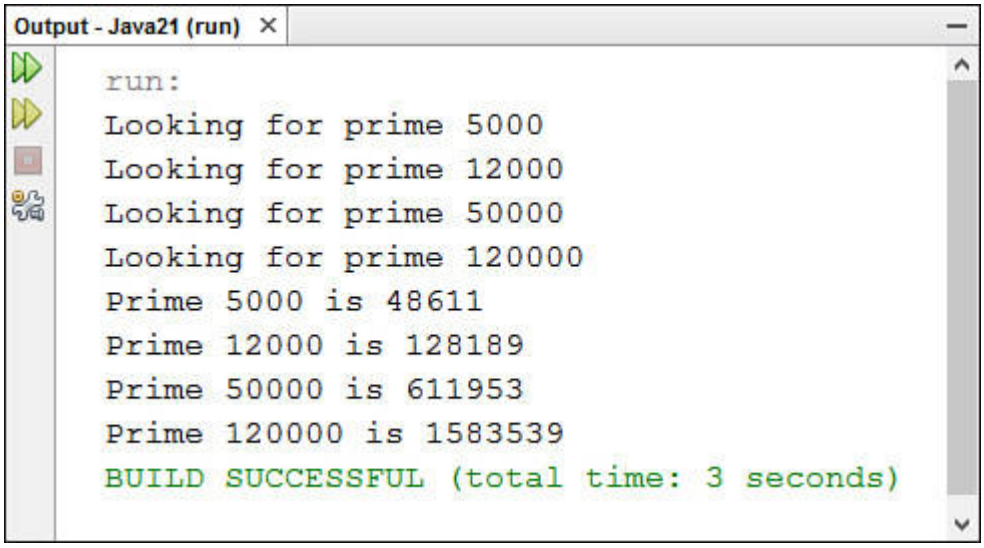


FIGURE 7.2 Using threads to find multiple primes in a sequence.

The `for` loop in lines 10–18 of the PrimeThreads application creates one `PrimeFinder` object for each command-line argument specified when the program is run.

Because arguments are `Strings` and the `PrimeFinder` constructor requires `long` values, the `Long.parseLong(String)` class method is used to handle the conversion. All the number-parsing methods throw `NumberFormatException` exceptions, so they are enclosed in `try-catch` blocks to deal with arguments that are not numeric.

When a `PrimeFinder` object is created, the object starts running in its own thread (as specified in the `PrimeFinder` constructor).

The `while` loop in lines 20–36 checks to see whether any `PrimeFinder` thread has completed, which is indicated by its `finished` instance variable equaling `true`. When a thread has completed, the `displayResult()` method is called in line 27 to display the prime number that was found. The thread then is set to `null`, freeing the object's resources (and preventing its result from being displayed more than once).

The call to `Thread.sleep(1000)` in line 32 causes the `while` loop to pause for one second during each pass through the loop. A slowdown in loops helps keep the JVM from executing statements at such a furious pace that it becomes bogged down.

Stopping a Thread

Stopping a thread is a little more complicated than starting one.

The best way to stop a thread is to place a loop in the thread's `run()` method that ends when a variable changes in value, as in the following example:

```
public void run() {
    while (okToRun == true) {
        // ...
    }
}
```

The `okToRun` variable could be an instance variable of the thread's class. If it is changed to `false`, the loop inside the `run()` method ends.

Another option you can use to stop a thread is to loop in the `run()` method only while the currently running thread has a variable that references it.

A class method, `Thread.currentThread()`, returns a reference to the current thread (in other words, the thread in which the object is running).

The following `run()` method loops as long as `runner` and `currentThread()` refer to the same object:

Click here to view code image

```
public void run() {
    Thread thisThread = Thread.currentThread();
    while (runner == thisThread) {
        // body of loop
    }
}
```

If you use a loop like this, you can stop the thread anywhere in the class with the following statement:

```
runner = null;
```

Summary

Exceptions and threads strengthen the robustness of your programs.

Exceptions enable you to manage potential errors. By using `try`, `catch`, and `finally`, you can protect code that might result in exceptions by handling those exceptions as they occur.

Handling exceptions is only half the battle; the other half is generating and throwing exceptions. A `throws` clause tells a method's users that the method might throw an exception. It also can be used to pass on an exception from a method call in the body of your method.

You learned how to create and throw your own methods by defining new exception classes and by throwing instances of any exception classes using `throw`.

Threads enable you to run the most processor-intensive parts of a Java class separately from the rest of the class. This is especially useful when the class is doing something computing-intensive such as animation, complex math, or looping through a large amount of data quickly.

You also can use threads to do several things at once and to start and stop threads externally.

Threads implement the `Runnable` interface, which contains one method: `run()`. When you start a thread by calling its `start()` method, the thread's `run()` method is called automatically.

Q&A

Q I'm still not sure I understand the difference between exceptions, errors, and runtime exceptions. Is there another way of looking at them?

A Errors are caused by dynamic linking or JVM problems. Thus, they are too low-level for most programs to care about—or to be able to handle even if they did care.

Runtime exceptions are generated by the normal execution of Java code. Although they occasionally reflect a condition you will want to handle explicitly, more often they reflect a coding mistake made by the programmer, and thus simply print an error to help flag that mistake.

Non-runtime exceptions (`IOException` exceptions, for example) are conditions that, because of their nature, should be explicitly handled by any robust and well-thought-out code. The Java Class Library has been written using only a few of these, but those few are important to using the system safely and correctly. The compiler helps you handle these exceptions properly via its `throws` clause checks and restrictions.

Q Does Java support unit testing to make programs more reliable?

A Unit testing, a technique for ensuring the reliability of software by adding tests, is supported by the open source Java Class Library JUnit. This is the most popular unit-testing framework for Java programmers. Visit www.junit.org to download it.

With JUnit, you write a set of tests, called a suite, that create the Java objects you've developed and call their methods. The values produced by these tests are checked to see whether they're what you expected. All tests must pass for your software to pass.

Although unit testing is only as good as the tests you create, the existence of a test suite is extremely helpful when you make changes to your software. By running the tests again after the changes, you can better assure yourself that it continues to work correctly.

Some Java programmers believe so strongly in the benefits of unit testing that they write tests before any code.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What keyword is used to jump out of a `try` block and into a `finally` block?

   A. `catch`

   B. `return`

   C. `while`

2. What class should be the superclass of any exceptions you create in Java?

   A. `Throwable`

   B. `Error`

   C. `Exception`

3. If a class implements the `Runnable` interface, what methods must the class contain?

   A. `start()`, `stop()`, and `run()`

   B. `actionPerformed()`

   C. `run()`

Answers

1. B. The `return` statement exits the block.

2. C. The kinds of errors you'll want to note in your programs generally belong in the `Exception` hierarchy.

3. C. The `Runnable` interface requires only the `run()` method.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

The AverageValue application is supposed to take up to 10 floating-point numbers as command-line arguments and display their average.

Given:

Click here to view code image

```
public class AverageValue {
    public static void main(String[] arguments) {
        float[] temps = new float[10];
        float sum = 0;
        int count = 0;
        int i;
        for (i = 0; i < arguments.length & i < 10; i++) {
            try {
                temps[i] = Float.parseFloat(arguments[i]);
                count++;
            } catch (NumberFormatException nfe) {
                System.out.println("Invalid input: " +
arguments[i]);
            }
            sum += temps[i];
        }
        System.out.println("Average: " + (sum / i));
    }
}
```

Which statement contains an error?

A. `for (i = 0; i < arguments.length & i < 10; i++) {`

B. `sum += temps[i];`

C. `System.out.println("Average: " + (sum / i));`

D. None of them; the program is correct.

The answer is available on the book's website at www.java21days.com. Visit the Day 7 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Modify the `PrimeFinder` class so that it throws a new exception, `NegativeNumberException`, if a negative number is sent to the constructor.
2. Modify the `PrimeThreads` application so that it can handle the new `NegativeNumberException` error.

Exercise solutions are offered on the book's website at www.java21days.com.

Week II: The Java Class Library

8 Data Structures

9 Working with Swing

10 Building a Swing Interface

11 Arranging Components on a User Interface

12 Responding to User Input

13 Creating Java2D Graphics

14 Developing Swing Applications

Day 8. Data Structures

During the first week, you learned about the core elements of the Java language: objects, classes, and interfaces, along with the keywords, statements, expressions, and operators they contain.

For the second week, the focus shifts from the classes you create to the ones that have been created for you. The Java Class Library is a set of standard packages from Oracle that has more than 4,200 classes you can use in your own Java programs.

Today, you start with classes that represent data.

The following data structures are covered:

- Bit sets, which hold Boolean values
- Array lists, arrays that can grow and shrink in size
- Stacks, structures stored in last-in, first-out (LIFO) order
- Hash maps, which store items using keys

Moving Beyond Arrays

The Java Class Library provides a set of data structures in the `java.util` package that gives you more flexibility in organizing and manipulating data.

A solid understanding of data structures and when to employ them will be useful throughout your Java programming efforts.

Many Java programs that you create rely on some means of storing and manipulating data within a class. Up to this point, you have used three structures to store and retrieve data: variables, `String` objects, and arrays.

These are just a few of the data classes available in Java. If you don't understand the full range of data structures, you'll find yourself trying to use arrays or strings when other options would be more efficient or easier to implement.

Outside of primitive types and strings, arrays are the simplest data structure that Java supports. An array is a series of data elements of the same primitive type or class. It's treated as a single object but contains multiple elements that can be accessed independently. Arrays are useful when you need to store and access related information.

A glaring limitation of arrays is that they can't adjust in size to accommodate more or fewer elements. You can't add new elements to an array that's already full. One data structure you learn about today, array lists, does not have this limitation.

Note

Unlike the data structures provided by the `java.util` package, arrays are considered such a core component of Java that they are implemented in the language itself. Therefore, you can use arrays in Java without using an object to hold their data.

Java Structures

The data structures provided by the `java.util` package perform a wide range of functions. These data structures consist of the `Iterator` interface, `Map` interface, and classes such as the following:

- `BitSet`
- `ArrayList`
- `Stack`
- `HashMap`

Each of these data structures provides a way to store and retrieve information in a well-defined manner. The `Iterator` interface itself isn't a data structure, but it defines a means to retrieve successive elements from a data structure. For example, `Iterator` defines a method called `next()` that gets the next element in a data structure containing multiple elements.

Note

`Iterator` is an expanded and improved version of the `Enumeration` interface from early versions of the language. Although `Enumeration` is still supported, `Iterator` should be used instead because it has simpler method names and support for removing items. `Iterator` also has been designed to detect a problem-prone situation with threads: It fails with a `ConcurrentModificationException` when one thread changes an item while another one is looping through the elements.

The `BitSet` class implements a group of bits, or flags, that can be set and cleared individually. This class is useful when you need to keep up with a set of Boolean values; you simply assign a bit to each value and set or clear it as appropriate. A flag is a Boolean value that represents one of a group of on/off type states in a program.

The `ArrayList` class is similar to an array, except that it can grow as necessary to accommodate new elements and also shrink. Like an array, elements of an `ArrayList` object can be accessed via an index value. The nice thing about using an array list is that you aren't required to give it a specific size upon creation; it shrinks and grows automatically as needed.

The `Stack` class implements a last-in, first-out stack of elements. You can think of a stack as a vertical stack of objects. When you add a new element, it's stacked on top of the others. When you pull an element off the stack, it comes off the top. The capability to remove an item differs from a structure like an array, where the elements always are available.

The `HashMap` class implements `Dictionary`, an abstract class that defines a data structure for mapping keys to values. This is useful when you want to access data through a particular key rather than an integer index. Because the `Dictionary` class is abstract, it provides only the framework for a key-mapped data structure rather than a specific implementation. A key is an identifier used to reference, or look up, a value in a data structure.

The `HashMap` class provides an implementation of a key-mapped data structure. `HashMap` organizes data based on a user-defined key structure. For example, in a ZIP Code list stored in a hash map, you could store data using each code as a key. The specific meaning of keys in a hash map depends on how the map is used and the data it contains.

The next section looks at these data structures in more detail to show how they work.

`Iterator`

The `Iterator` interface provides a standard means of progressing through a list of elements in a defined sequence, which is a common task for many data structures.

Even though you can't use the interface outside a particular data structure, understanding how the `Iterator` interface works helps you understand

other Java data structures.

With that in mind, take a look at three methods defined by the `Iterator` interface:

```
public boolean hasNext();

public Object next();

public void remove();
```

These methods lack code because interfaces don't have implementations. The class that implements the interface must provide the code to define the methods.

The `hasNext()` method determines whether the structure contains any more elements. You can call this method to see whether you can continue iterating through a structure.

The `next()` method retrieves the next element in a structure. If there are no more elements, `next()` throws a `NoSuchElementException` exception. To avoid this, you can use `hasNext()` in conjunction with `next()` to make sure that there is another element to retrieve.

The following `while` loop uses these two methods to iterate through a data structure called `users` that implements the `Iterator` interface:

```
while (users.hasNext()) {
    Object ob = users.next();
    System.out.println(ob);
}
```

This sample code displays the contents of each list item by using the `hasNext()` and `next()` methods.

The `next()` method returns an object of the class `Object`. You can cast this to another class that the structure holds. Here's an example for a data structure that holds `String` objects:

Click here to view code image

```
while (users.hasNext()) {
    String ob = (String) users.next();
    System.out.println(ob);
}
```

Note

Because `Iterator` is an interface, you never use it directly as a data structure. Instead, you use the methods defined by `Iterator` for structures that implement the interface. This provides a consistent way to work with many of Java's standard data structures, which makes them easier to learn and use.

Bit Sets

The `BitSet` class is useful when you need to represent a large amount of binary data—bit values that equal either 0 or 1. These also are called on-or-off values (with 1 representing on and 0 representing off) or Boolean values (with 1 `true` and 0 `false`).

With the `BitSet` class, you can use individual bits to store Boolean values without requiring bitwise operations to extract bit values. You simply refer to each bit using an index. Another nice feature of `BitSet` is that it automatically grows to represent the number of bits that a program requires. Figure 8.1 shows the logical organization of a bit set data structure.

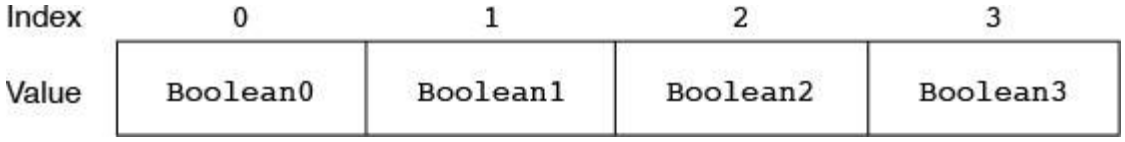


FIGURE 8.1 The organization of a bit set.

You can use a `BitSet` object to hold attributes that easily can be modeled by Boolean values. Because the individual bits in a set are accessed via an index, you can define each attribute as a constant index value, as in this class:

Click here to view code image

```
class ConnectionAttributes {
    public static final int READABLE = 0;
    public static final int WRITABLE = 1;
    public static final int STREAMABLE = 2;
    public static final int FLEXIBLE = 3;
}
```

In this class, the attributes are assigned increasing values beginning with 0. You can use these values to get and set the appropriate bits in a set. First, you need to create a `BitSet` object:

```
BitSet connex = new BitSet();
```

This constructor creates a set with no specified size. You also can create a set with a specific size:

```
BitSet connex = new BitSet(4);
```

This creates a set containing four Boolean bits. Regardless of the constructor used, all bits in new sets initially are set to `false`. After you have a set, you can set and clear the bits by using `set(int)` and `clear(int)` methods with the bit constants you defined:

Click here to view code image

```
connex.set(ConnectionAttributes.WRITABLE);
connex.set(ConnectionAttributes.STREAMABLE);
connex.set(ConnectionAttributes.FLEXIBLE);

connex.clear(ConnectionAttributes.WRITABLE);
```

In this code, the `WRITABLE`, `STREAMABLE`, and `FLEXIBLE` attributes are set, and then the `WRITABLE` bit is cleared. The class name is used for each attribute because the constants are class variables in the `ConnectionAttributes` class.

You can get the value of individual bits in a set by using the `get()` method:

Click here to view code image

```
boolean isWriteable = connex.get(ConnectionAttributes.WRITABLE);
```

You can find out how many bits a set represents with the `size` method:

```
int numBits = connex.size();
```

The `BitSet` class also provides other methods for performing comparisons and bitwise operations on sets, such as `AND`, `OR`, and `XOR`. All these methods take a `BitSet` object as their only argument.

Today's first project is `HolidaySked`, a Java class that uses a set to keep track of which days in a year are holidays.

A set is employed because `HolidaySked` must be able to take any day of the year and answer the same yes/no question: Are you a holiday?

Enter the code shown in Listing 8.1 into an empty Java file in NetBeans named `HolidaySked` in the `com.java21days` package.

LISTING 8.1 The Full Text of `HolidaySked.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class HolidaySked {
 6:     BitSet sked;
 7:
 8:     public HolidaySked() {
 9:         sked = new BitSet(365);
10:         int[] holiday = { 1, 15, 50, 148, 185, 246,
11:             281, 316, 326, 359 };
12:         for (int i = 0; i < holiday.length; i++) {
13:             addHoliday(holiday[i]);
14:         }
15:     }
16:
17:     public void addHoliday(int dayToAdd) {
18:         sked.set(dayToAdd);
19:     }
20:
21:     public boolean isHoliday(int dayToCheck) {
22:         boolean result = sked.get(dayToCheck);
23:         return result;
24:     }
25:
26:     public static void main(String[] arguments) {
27:         HolidaySked cal = new HolidaySked();
28:         if (arguments.length > 0) {
29:             try {
30:                 int whichDay =
Integer.parseInt(arguments[0]);
31:                 if (cal.isHoliday(whichDay)) {
32:                     System.out.println("Day number " +
whichDay +
33:                         " is a holiday.");
34:                 } else {
35:                     System.out.println("Day number " +
whichDay +
36:                         " is not a holiday.");
37:                 }
38:             } catch (NumberFormatException nfe) {
39:                 System.out.println("Error: " +
nfe.getMessage());
40:             }
41:         }
```

```
42:     }
43: }
```

This application requires one command-line argument: a number from 1 to 365 that represents the day of the year, in sequence. (These numbers are defined in lines 10–11 and would be different for each year.) Use the command Run, Set Project Configuration, Customize to set the argument.

Test the program with values such as 15 (Martin Luther King Day) or 103 (my birthday). The application should respond that day 15 is a holiday but that day 103, sadly, is not.

The output of the application for day 170 is shown in Figure 8.2.

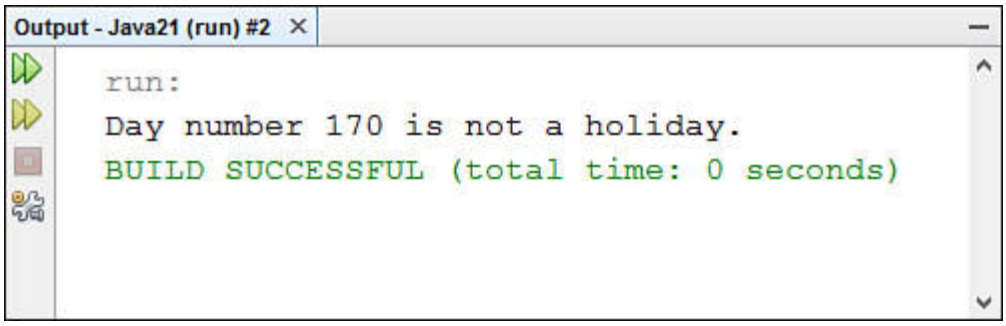


FIGURE 8.2 Trying out the `BitSet` data structure.

The `HolidaySked` class contains only one instance variable: `sked`, a `BitSet` that holds values for each day in a year.

The constructor of the class creates the `sked` bit set with 365 positions, with a value of 0 (lines 8–15). All bit sets are filled with 0 values when they are created.

Next, an integer array called `holiday` is created. This array holds the number of each work holiday in the year, beginning with 1 (New Year's Day) and ending with 359 (Christmas).

The `holiday` array is used to add each holiday to the `sked` bit set. A `for` loop iterates through the `holiday` array and calls the method `addHoliday(int)` with each one (lines 12–14).

The `addHoliday(int)` method is defined in lines 17–19. The argument represents the day that should be added. The bit set's `set(int)` method is called to set the bit at the specified position to 1. For example, if `set(359)` is called, the bit at position 359 is given the value 1.

The `HolidaySked` class also can determine whether a specified day is a holiday. This is handled by the `isHoliday(int)` method (lines 21–24). The method calls the bit set's `get(int)` method, which returns `true` if the specified position has the value 1 and `false` otherwise.

This class can be run as an application because of the `main()` method (lines 26–42). The application takes a single command-line argument: a number from 1 to 365 that represents one of the days of the year. The application displays whether that day is a holiday according to the schedule of the `HolidaySked` class.

Array Lists

One of the most popular data structures in Java, the `ArrayList` class implements an expandable and contractible array of objects, making it more flexible and useful than arrays. Because the `ArrayList` class is responsible for changing size as necessary, it has to decide when and how much to grow or shrink as elements are added and removed.

An array list can be created with a constructor taking no arguments:

Click here to view code image

```
ArrayList golfer = new ArrayList();
```

This constructor creates a default array list containing no elements. All lists are empty upon creation. One of the attributes that determines how a list sizes itself is its initial capacity—the number of elements for which it allocates memory to hold.

The size of an array list is the number of elements currently stored in it. A list's capacity is always greater than or equal to the size.

The following code shows how to create an array list with a specified capacity:

Click here to view code image

```
ArrayList golfer = new ArrayList(30);
```

This list allocates enough memory to support 30 elements. If the capacity fills up, the list automatically expands by half the initial size. So if a 30th element is put in `golfer`, it expands to make room for 45 elements.

Because allocating additional space for the list takes time and consumes memory, it's best to create a list with as many elements as you expect to use.

You can't just use square brackets `[]` to access the elements in an array list, as you can in an array. You must use methods of the `ArrayList` class.

Use the `add(Object)` method to add an element to an array list, like this:

```
golfer.add("Park");
golfer.add("Lewis");
golfer.add("Ko");
```

The `lastElement()` method returns an `Object` because the `ArrayList` class supports all classes of objects. You must cast it to the class that was put into the list. Here, because strings were stored in `golfer`, the returned object is cast to a string.

The `get()` method retrieves a list element using a numeric index, as shown in the following code:

Click here to view code image

```
String s1 = (String) golfer.get(0);
String s2 = (String) golfer.get(2);
```

Because array list numbering is zero-based, the first call to `get()` retrieves the "Park" string, and the second call retrieves the "Lewis" string.

Just as you can retrieve an element at a particular index, you also can add and remove elements at an index by using the `add(int, Object)` and `remove(int)` methods:

```
golfer.add(1, "Kim");
golfer.add(0, "Thompson");
golfer.remove(3);
```

The first call to `add()` inserts an element at index 1, between the "Park" and "Lewis" strings. The "Lewis" and "Ko" strings are moved by an element in the list to accommodate the inserted "Kim" string. The second call to `add()` inserts an element at index 0, which is the beginning of the list. All existing elements are moved up one space in the list to accommodate the inserted "Thompson" string. At this point, the contents of the list look like this:

0. "Thompson"

1. "Park"

2. "Kim"

3. "Lewis"

4. “Ko”

The call to `remove()` removes the element at index 3, which is the “Lewis” string. The resulting list consists of the following strings:

0. “Thompson”
1. “Park”
2. “Kim”
3. “Ko”

You can use the `set()` method to change a specific element:

```
golfer.set(1, "Pressel");
```

This method replaces the “Park” string with the “Pressel” string, resulting in the following list:

0. “Thompson”
1. “Pressel”
2. “Kim”
3. “Ko”

If you want to clear out the array list, you can remove all the elements with the `clear()` method:

```
golfer.clear();
```

The `ArrayList` class also provides some methods for working with elements without using indexes. These methods search through the list for a particular element. The first of these methods is the `contains(Object)` method, which simply checks whether an object is in the list:

Click here to view code image

```
boolean isThere = golfer.contains("Kerr");
```

Another method for searching is the `indexOf(Object)` method, which finds the index of an element matching an object:

```
int i = golfer.indexOf("Ko");
```

The `indexOf()` method returns the index or –1 if the object is not in the list. The `remove(Object)` method works similarly, removing an object from the list, as in this statement:

```
golfer.remove("Pressel");
```

The `ArrayList` class offers a few methods for determining and manipulating a list's size. First, the `size` method determines the number of elements in the list:

```
int size = golfer.size();
```

Recall that lists have two attributes relating to size: size and capacity. The size is the number of elements in the list, and the capacity is the amount of memory allocated to hold all the elements. The capacity always is greater than or equal to the size. You can force the capacity to exactly match the size by using the `trimToSize()` method:

```
golfer.trimToSize();
```

Caution

The Java Class Library also includes `Vector`, a data structure that works a lot like array lists. When you use vectors in NetBeans, a warning is displayed that calls the class an "obsolete collection." This occurs because array lists are considered a superior version of vectors.

Looping Through Data Structures

If you're interested in working sequentially with all the elements in a list, you can use the `iterator()` method, which returns an `Iterator` that holds a list of the elements you can loop through:

Click here to view code image

```
Iterator it = golfer.iterator();
```

As you learned earlier today, you can use an iterator to step through elements sequentially. In this example, you can work with the `it` list using the methods defined by the `Iterator` interface.

The following `for` loop uses an iterator and its methods to traverse an entire array list:

Click here to view code image

```
for (Iterator i = golfer.iterator(); i.hasNext(); ) {
    String name = (String) i.next();
```

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

Today's next project demonstrates the care and feeding of array lists. The `CodeKeeper` class, shown in Listing 8.2, holds a set of text codes, some provided by the class and others provided by users. Because the amount of space needed to hold the codes isn't known until the program is run, an array list is used to store the data instead of an array. Create this class in NetBeans, remembering to put it in the `com.java21days` package.

LISTING 8.2 The Full Text of `CodeKeeper.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class CodeKeeper {
 6:     ArrayList list;
 7:     String[] codes = { "alpha", "lambda", "gamma", "delta",
"zeta" };
 8:
 9:     public CodeKeeper(String[] userCodes) {
10:         list = new ArrayList();
11:         // load built-in codes
12:         for (int i = 0; i < codes.length; i++) {
13:             addCode(codes[i]);
14:         }
15:         // load user codes
16:         for (int j = 0; j < userCodes.length; j++) {
17:             addCode(userCodes[j]);
18:         }
19:         // display all codes
20:         for (Iterator ite = list.iterator(); ite.hasNext();
) {
21:             String output = (String) ite.next();
22:             System.out.println(output);
23:        }
24:     }
25:
26:     private void addCode(String code) {
27:         if (!list.contains(code)) {
28:             list.add(code);
29:         }
30:     }
31:
```

```
32:     public static void main(String[] arguments) {
33:         CodeKeeper keeper = new CodeKeeper(arguments);
34:     }
35: }
```

NetBeans may display a warning that this class uses “unchecked or unsafe operations.” This isn’t as severe as it sounds. The code works properly as written and is not unsafe.

The warning serves as a strong hint that there’s a better way to work with array lists and other data structures. You’ll learn about this technique later today.

The `CodeKeeper` class uses an `ArrayList` instance variable named `list` to hold the text codes.

First, five built-in codes are read from a string array into the list (lines 12–14).

Next, any codes provided by the user as command-line arguments are added (lines 16–18).

Codes are added by calling the `addCode()` method (lines 26–30). `addCode()` adds a new text code only if it isn’t already present, using the list’s `contains(Object)` method to make this determination.

You add command-line arguments in NetBeans by selecting Project, Set Project Configuration, Customize. The arguments should be a list of codes separated by spaces.

After the codes have been added to the list, its contents are displayed. Running the class with the command-line arguments “beta” and “epsilon” produces the output shown in Figure 8.3.

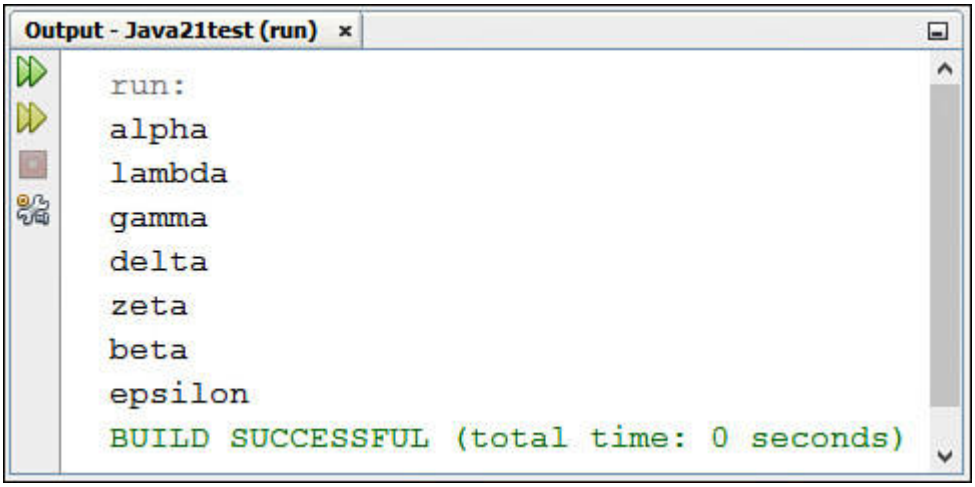


FIGURE 8.3 Manipulating and displaying an array list.

A simpler `for` loop can be employed to iterate through a data structure. The loop takes the form `for (`*`variable`*` : `*`structure`*`)`, where *`structure`* is a data structure that implements the `Iterator` interface. The *`variable`* section declares an object that holds each element of the structure as the loop progresses.

This `for` loop uses an iterator and its methods to traverse an array list named `golfer`:

```
for (Object name : golfer) {
    System.out.println(name);
}
```

The loop can be used with any data structure that works with `Iterator`.

Stacks

Stacks are a data structure used to model information accessed in a specific order. The `Stack` class in Java is implemented as a last-in, first-out stack, which means that the last item added to the stack is the first one to be removed. Figure 8.4 shows the logical organization of a stack.

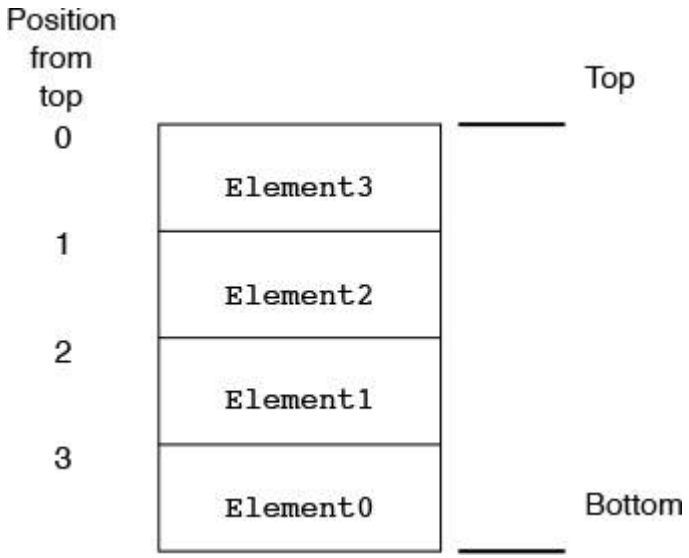


FIGURE 8.4 The organization of a stack.

You might wonder why the numbers of the elements don't match their positions from the top of the stack. Keep in mind that elements are added to the top, so `Element0`, which is on the bottom, was the first element added to the stack. Likewise, `Element3`, which is on top, was the last element added. Also, because `Element3` is at the top of the stack, it will be the first to be removed.

The `Stack` class defines only one constructor, which is a default constructor that creates an empty stack. You use this constructor to create a stack like this:

```
Stack s = new Stack();
```

Stacks in Java contain methods to manipulate the stack.

You can add new elements to a stack by using the `push()` method, which pushes an element onto the top of the stack:

```
s.push("One");
s.push("Two");
s.push("Three");
s.push("Four");
s.push("Five");
s.push("Six");
```

This code pushes six strings onto the stack, with the last string ("Six") ending up on top. You remove elements from the stack by using the `pop()` method, which pops them off the top:

```
String s1 = (String) s.pop();
String s2 = (String) s.pop();
```

This code pops the last two strings off the stack, leaving the first four strings. This code results in the `s1` variable's containing the "Six" string and the `s2` variable's containing the "Five" string.

If you want to use the top element on the stack without actually popping it off the stack, you can use the `peek()` method:

```
String s3 = (String) s.peek();
```

This call to `peek()` returns the "Four" string but leaves the string on the stack. You can search for an element on the stack by using the `search()` method:

```
int i = s.search("Two");
```

The `search()` method returns the distance from the top of the stack to the element if it is found, or –1 if not. In this case, the "Two" string is the third element from the top, so the `search()` method returns 2.

Note

As in all Java data structures that deal with indexes or lists, the `Stack` class reports element positions in a zero-based fashion: The top element in a stack has a location of 0, the fourth element down has a location of 3, and so on.

The last method defined in the `Stack` class is `empty()`, which indicates whether a stack is empty:

```
boolean isEmpty = s.empty();
```

Map

The `Map` interface defines a framework for implementing a key-mapped data structure, a place to store objects each referenced by a key. The key serves the same purpose as an element number in an array—it's a unique value used to access the data stored at a position in the data structure.

You can put the key-mapped approach to work by using the `HashMap` class or one of the other classes that implement the `Map` interface. You learn about the `HashMap` class in the next section.

The `Map` interface defines a means of storing and retrieving information based on a key. This is similar in some ways to the `ArrayList` class, in which elements are accessed through an index, which is a specific type of key. However, keys in the `Map` interface can be just about anything. You can create your own classes to use as the keys for accessing and manipulating data in a dictionary. Figure 8.5 shows how keys map to data in a dictionary.

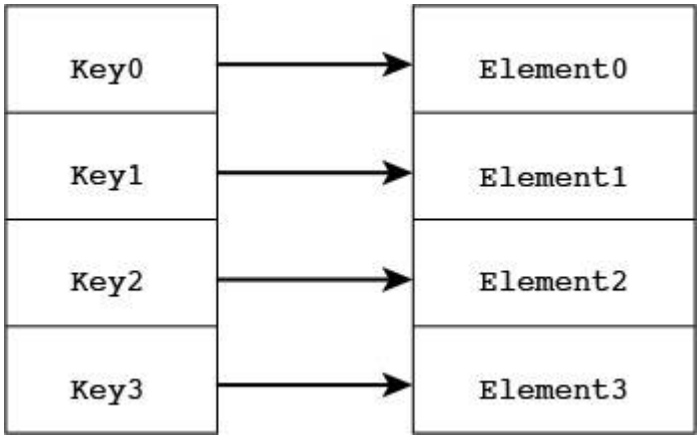


FIGURE 8.5 The organization of a key-mapped data structure.

The `Map` interface declares a variety of methods for working with the data stored in a dictionary. Implementing classes have to implement all those methods to be truly useful. The `put(String, Object)` and `get(String, Object)` methods are used to store objects in the dictionary and retrieve them.

Assuming that `look` is an object that implements the `Map` interface, the following code shows how to use the `put()` method to add elements:

Click here to view code image

```
Rectangle r1 = new Rectangle(0, 0, 5, 5);
look.put("small", r1);
Rectangle r2 = new Rectangle(0, 0, 15, 15);
look.put("medium", r2);
Rectangle r3 = new Rectangle(0, 0, 25, 25);
look.put("large", r3);
```

This code adds three `Rectangle` objects to the map (from the `java.awt` package), using strings as the keys. To get an element, use the `get()` method and specify the appropriate key:

Click here to view code image

```
Rectangle r = (Rectangle) look.get("medium");
```

You also can remove an element with a key by using the `remove()` method:

```
look.remove("large");
```

You can find out how many elements are in the structure by using the `size()` method, as in the `ArrayList` class:

```
int size = look.size();
```

You also can check whether the structure is empty by using the `isEmpty()` method:

Click here to view code image

```
boolean isEmpty = look.isEmpty();
```

Hash Maps

The `HashMap` class implements the `Map` interface and provides a complete implementation of a key-mapped data structure. Hash maps let you store data based on some type of key and have an efficiency defined by the map's load factor. The load factor is a floating-point number between 0.0 and 1.0 that determines how and when the hash map allocates space for more elements.

Like array lists, hash maps have a capacity, or an amount of allocated memory. Hash maps allocate memory by comparing the map's current size with the product of the capacity and the load factor. If the size of the hash map exceeds this product, the map increases its capacity by rehashing itself.

Load factors closer to 1.0 result in a more efficient use of memory at the expense of a longer lookup time for each element. Similarly, load factors closer to 0.0 result in more efficient lookups but tend to be more wasteful with memory. Determining the load factor for your own hash maps depends on how you use each map and whether your priority is performance or memory efficiency.

You can create hash maps in one of three ways. The first constructor creates a default hash map with an initial capacity of 16 elements and a load factor of 0.75:

```
HashMap hash = new HashMap();
```

The second constructor creates a hash map with the specified initial capacity and a load factor of 0.75:

Click here to view code image

```
HashMap hash = new HashMap(20);
```

Finally, the third constructor creates a hash map with the specified initial capacity and load factor:

Click here to view code image

```
HashMap hash = new HashMap(20, 0.5F);
```

All the abstract methods defined in `Map` are implemented in the `HashMap` class. In addition, the `HashMap` class implements a few others that perform functions specific to supporting maps. One of these is the `clear()` method, which clears a map of all its keys and elements:

```
hash.clear();
```

The `containsValue(Object)` method checks whether an object is stored in the hash map:

Click here to view code image

```
Rectangle box = new Rectangle(0, 0, 5, 5);
boolean isThere = hash.containsValue(box);
```

The `containsKey(String)` method searches a map for a key:

Click here to view code image

```
boolean isThere = hash.containsKey("Small");
```

The practical use of a hash map comes from its capability to represent data that is too time-consuming to search or reference by value. The data structure comes in handy when you're working with complex data and it's more efficient to access the data by using a key rather than comparing the data objects themselves.

This key, which is called a hash code, is a computed key that uniquely identifies each element in a hash map.

This technique of computing and using hash codes for object storage and reference is exploited heavily throughout the Java Class Library. The parent of all classes, `Object`, defines a `hashCode()` method overridden in most standard Java classes. Any class that defines a `hashCode()` method can be efficiently stored and accessed in a hash map. A class that wants to be hashed also must implement the `equals()` method, which defines a way of telling whether two objects are equal. The `equals()` method usually just

performs a straight comparison of all the member variables defined in a class.

The next project you undertake today uses maps for a shopping application.

The ComicBooks application prices collectible comic books according to their base value and condition. The condition is described as one of the following: mint, near mint, very fine, fine, good, or poor. Each condition has a specific effect on a comic's value:

- "Mint" books are worth 3 times their base price.
- "Near mint' books are worth 2 times their base price.
- "Very fine" books are worth 1.5 times their base price.
- "Fine" books are worth their base price.
- "Good" books are worth 0.5 times their base price.
- "Poor" books are worth 0.25 times their base price.

To associate text such as "mint" or "very fine" with a numeric value, they are put into a hash map. The keys to the map are the condition descriptions, and the values are floating-point numbers such as 3.0, 1.5, and 0.25.

Enter the code shown in Listing 8.3 in NetBeans as the class `ComicBooks` in the package `com.java21days`.

LISTING 8.3 The Full Text of `ComicBooks.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class ComicBooks {
 6:
 7:     public ComicBooks() {
 8:     }
 9:
10:     public static void main(String[] arguments) {
11:         // set up hash map
12:         HashMap quality = new HashMap();
13:         float price1 = 3.00F;
14:         quality.put("mint", price1);
15:         float price2 = 2.00F;
16:         quality.put("near mint", price2);
```

```
17:          float price3 = 1.50F;
18:          quality.put("very fine", price3);
19:          float price4 = 1.00F;
20:          quality.put("fine", price4);
21:          float price5 = 0.50F;
22:          quality.put("good", price5);
23:          float price6 = 0.25F;
24:          quality.put("poor", price6);
25:          // set up collection
26:          Comic[] comix = new Comic[3];
27:          comix[0] = new Comic("Amazing Spider-Man", "1A",
"very fine",
28:              12_000.00F);
29:          comix[0].setPrice( (Float)
quality.get(comix[0].condition) );
30:          comix[1] = new Comic("Incredible Hulk", "181", "near
mint",
31:              680.00F);
32:          comix[1].setPrice( (Float)
quality.get(comix[1].condition) );
33:          comix[2] = new Comic("Cerebus", "1A", "good",
190.00F);
34:          comix[2].setPrice( (Float)
quality.get(comix[2].condition) );
35:          for (int i = 0; i < comix.length; i++) {
36:              System.out.println("Title: " + comix[i].title);
37:              System.out.println("Issue: " +
comix[i].issueNumber);
38:              System.out.println("Condition: " +
comix[i].condition);
39:              System.out.println("Price: $" + comix[i].price +
"\n");
40:          }
41:      }
42: }
43:
44: class Comic {
45:      String title;
46:      String issueNumber;
47:      String condition;
48:      float basePrice;
49:      float price;
50:
51:      Comic(String inTitle, String inIssueNumber, String
inCondition,
52:          float inBasePrice) {
53:
54:          title = inTitle;
55:          issueNumber = inIssueNumber;
56:          condition = inCondition;
```

```
57:          basePrice = inBasePrice;
58:      }
59:
60:      void setPrice(float factor) {
61:          price = basePrice * factor;
62:      }
63: }
```

When you run the ComicBooks application, it produces the output in Figure 8.6.

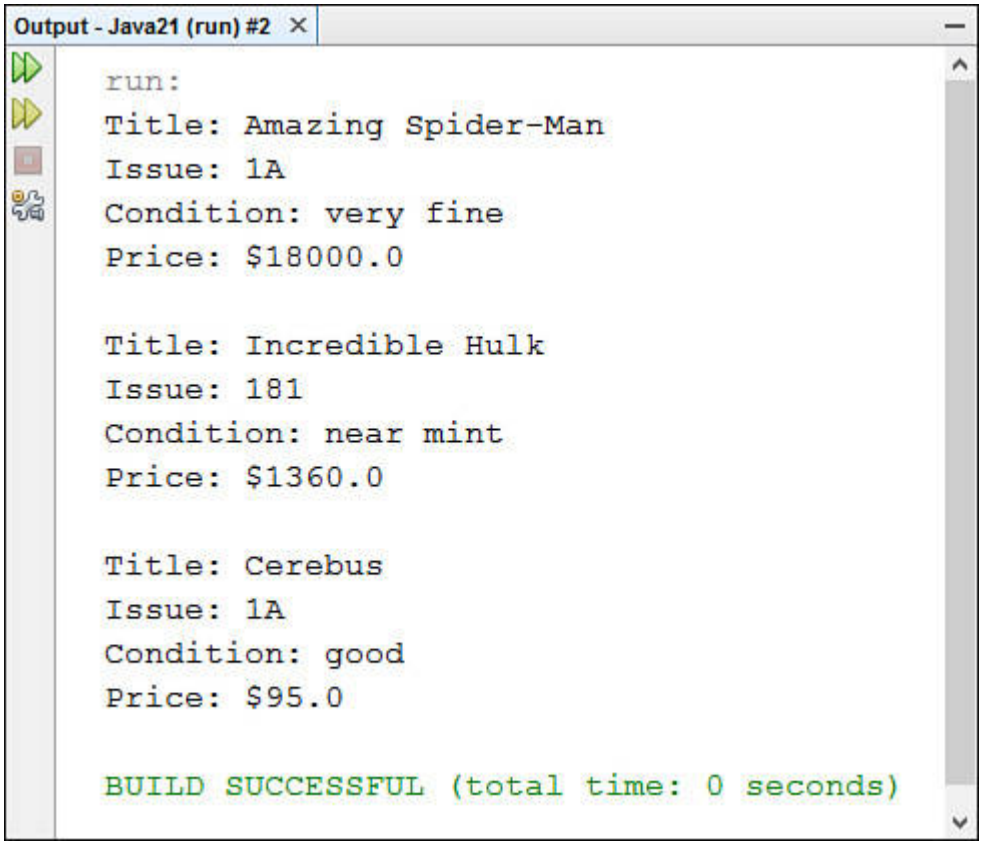


FIGURE 8.6 Storing comic book values in a hash map.

The ComicBooks application is implemented as two classes: an application class called `ComicBooks` and a helper class called `Comic`.

In the application, the hash map is created in lines 12–24.

First, the map is created in line 12.

Next, a `float` called `price1` is created with the value 3.00. This value is added to the map and associated with the key "mint". (Remember that hash maps, like other data structures, can hold only objects. The float value is automatically converted to a `Float` object through autoboxing.)

The process is repeated for each of the other comic book conditions, from "near mint" to "poor."

After the hash map is set up, an array of `Comic` objects called `comix` is created to hold each comic book currently for sale.

The `Comic` constructor is called with four arguments: the book's title, issue number, condition, and base price. The first three are strings, and the last is a `float`.

After a `Comic` has been created, its `setPrice(float)` method is called to set the book's price based on its condition. Here's an example, line 29:

Click here to view code image

```
comix[0].setPrice( (Float) quality.get(comix[0].condition) );
```

The hash map's `get(String)` method is called with the book's condition, a string that is one of the keys in the map. An `Object` is returned that represents the value associated with that key. (In line 29, because `comix[0].condition` is equal to "very fine", `get()` returns the floating-point value 3.00F.)

Because `get()` returns an `Object`, it must be cast as a `Float`. The `Float` argument is unboxed as a `float` value automatically through unboxing.

This process is repeated for two more books.

Lines 35–40 display information about each comic book in the `comix` array.

The `Comic` class is defined in lines 44–63. It has five instance variables—the `String` object's `title`, `issueNumber`, and `condition`, and the floating-point value's `basePrice` and `price`.

The constructor method of the class, located in lines 51–58, sets the value of four instance variables to the arguments sent to the constructor.

The `setPrice(Float)` method in lines 60–62 sets the price of a comic book. The argument sent to the method is a `float` value. A comic's price is calculated by multiplying this `float` by the comic's base price. Consequently, if a book is worth $1,000, and its multiplier is 2.0, the book is priced at $2,000.

Hash maps are a powerful data structure for manipulating large amounts of data. The fact that these maps are so widely supported in the Java Class

Library via the `Object` class should give you a clue as to their importance in Java programming.

Generics

The data structures that you learned about today are some of the most essential utility classes in the Java Class Library.

Hash maps, array lists, stacks, and the other structures in the `java.util` package are useful regardless of the kind of programs you want to develop. Almost every software program handles data in some manner.

These data structures are well-suited for use in code that applies generically to a wide range of classes of objects. A method written to manipulate array lists could be written to function equally well on strings, string buffers, character arrays, or other objects that represent text. A method in an accounting program could take objects that represent integers, floating-point numbers, and other math classes, using each to calculate a balance.

This flexibility comes at a price: When a data structure works with any kind of object, the Java compiler can't display a warning when the structure is being misused.

For instance, the ComicBooks application uses a hash map named `quality` to associate condition descriptions such as "mint" and "good" with price multipliers. Here's the statement for "near mint":

Click here to view code image

```
quality.put("near mint", 1.50F);
```

By design, the `quality` map should hold only floating-point values (as `Float` objects). However, the class compiles successfully regardless of the class of the value added to a map. You might goof and unintentionally add a string to the map, as in this revised statement:

Click here to view code image

```
quality.put("near mint", "1.50");
```

The class compiles successfully, but when it is run, it fails with a `ClassCastException` error in the following statement:

Click here to view code image

```
comix[1].setPrice( (Float) quality.get(comix[1].condition) );
```

The reason for the error is that the statement tries to cast the map's "near mint" value to a `Float`, which fails because it receives the string "1.50" instead.

Runtime errors are much more troublesome for programmers than compiler errors. A compiler error stops you in your tracks and must be fixed before you can continue. A runtime error might creep its way into the code, unbeknownst to you, and cause problems for users of your software.

You can specify the class or classes expected in a data structure using a feature of the language called generics.

The expected class information is added to statements where the structure is assigned a variable or created with a constructor. The class or classes are placed within < and > characters and follow the name of the class, as in this statement:

Click here to view code image

```
ArrayList<Integer> zipCodes = new ArrayList<>();
```

This statement creates an array list that will be used to hold `Integer` objects. The compiler uses inference to correctly guess the type of the class the second time the < and > characters appear. The <> after a class name sometimes is called a *diamond operator*. Here's another example:

Click here to view code image

```
HashMap<String, Float> quality = new HashMap<String, Float>();
```

The diamond operator <> infers the classes based on what they would have to be for the statement to make sense.

Because the list is declared with a class specified, the following statements cause a compiler error that NetBeans will flag in the source code editor:

```
zipCodes.add("90210");
zipCodes.add("02134");
zipCodes.add("20500");
```

The compiler recognizes that `String` objects do not belong in this array list. The proper way to add elements to the list is to use integer values:

```
zipCodes.add(90210);
zipCodes.add(02134);
zipCodes.add(20500);
```

These integers are converted to `Integer` objects by autoboxing.

Data structures that use multiple classes, such as hash maps, take these class names separated by commas within the < and > characters.

The ComicBooks application can take advantage of generics by changing line 10 of Listing 8.3 to the following:

Click here to view code image

```
HashMap<String, Float> quality = new HashMap<>();
```

This sets up a map to use `String` objects for keys and `Float` objects for values. With this statement in place, a string no longer can be added as the value for a condition such as "near mint." A compiler error flags a problem of this kind.

Generics also make it easier to retrieve an object from a data structure, because you don't have to use casting to convert them to the desired class. For example, the `quality` map no longer requires a cast to produce `Float` objects in statements like this one:

Click here to view code image

```
comix[1].setPrice(quality.get(comix[1].condition));
```

From a stylistic standpoint, the addition of generics in variable declarations and constructor methods is likely to appear intimidating. However, after you become accustomed to working with them (and using autoboxing, unboxing, and the new `for` loops), data structures are significantly easier to work with and less error-prone.

The `CodeKeeper2` class, shown in Listing 8.4, is a new version of `CodeKeeper` that has been rewritten to use generics, type inference, and the `for` loop that can iterate through data structures such as array lists.

LISTING 8.4 The Full Text of `CodeKeeper2.java`

Click here to view code image

```
1: package com.java21days;
2:
3: import java.util.*;
4:
5: public class CodeKeeper2 {
6:     ArrayList<String> list;
```

```
 7:     String[] codes = { "alpha", "lambda", "gamma", "delta",
"zeta" };
 8:
 9:     public CodeKeeper2(String[] userCodes) {
10:         list = new ArrayList<>();
11:         // load built-in codes
12:         for (int i = 0; i < codes.length; i++) {
13:             addCode(codes[i]);
14:         }
15:         // load user codes
16:         for (int j = 0; j < userCodes.length; j++) {
17:             addCode(userCodes[j]);
18:         }
19:         // display all codes
20:         for (String code : list) {
21:             System.out.println(code);
22:         }
23:     }
24:
25:     private void addCode(String code) {
26:         if (!list.contains(code)) {
27:             list.add(code);
28:         }
29:     }
30:
31:     public static void main(String[] arguments) {
32:         CodeKeeper2 keeper = new CodeKeeper2(arguments);
33:     }
34: }
```

The only modifications to the class are in line 6, where the generics declaration for an array list of strings is made; line 10, where type inference figures out the proper generics declaration; and lines 20–22, the simpler `for` loop that displays all the codes.

Enumerations

A common use of constants in Java is to attach a meaningful label to a series of integers, which you did earlier today as you worked with bit sets:

Click here to view code image

```
class ConnectionAttributes {
    public static final int READABLE = 0;
    public static final int WRITABLE = 1;
    public static final int STREAMABLE = 2;
    public static final int FLEXIBLE = 3;
}
```

These constants are useful because of the extra information provided in statements that contain them. Compare these two statements, which do the same thing:

Click here to view code image

```
setConnectionType(1);

setConnectionType(ConnectionAttributes.WRITABLE);
```

The latter is much easier to understand for a programmer examining the code.

Java has a data type called enumerations that serve the same purpose and have advantages over using constants in a class. The `enum` keyword is used in place of `class` and the values are separated by commas.

Here's a simple enumeration called Compass for the eight compass directions:

```
public enum Compass {
    NORTH,
    EAST,
    SOUTH,
    WEST,
    NORTHEAST,
    SOUTHEAST,
    SOUTHWEST,
    NORTHWEST
}
```

Each of these values is implicitly `static` and `final`, just like constants. They can appear in statements, method calls, and other code just like they were class constants. Here's an application that uses the enumeration:

Click here to view code image

```
public class DirectionSetter {
    Compass current;
    public void setDirection(Compass dir) {
        current = dir;
    }

    public static void main(String[] arguments) {
        DirectionSetter app = new DirectionSetter();
        app.setDirection(Compass.WEST);
        System.out.println(app.current);
    }
}
```

This class sets the current instance variable to `WEST` from the `Compass` enumeration and displays the variable, which is output as the text "WEST".

An advantage to using `enum` over class constants is that the compiler can detect errors when an invalid value is used. The only acceptable values that can be sent to the `setDirection(Compass)` method are the values of the `Compass` enumeration.

By comparison, a method that took `ConnectionAttributes` values as an argument could be called with any integer value.

There are other advantages to enumerations, which can function like a class with methods and variables of their own.

Any time you need a fixed set of constants, you can make them an enumeration.

Summary

Today you learned about several data structures you can use in your Java programs:

- **Bit sets**—Large sets of Boolean on-or-off values
- **Array lists**—Arrays that can change in size dynamically and be shrunken or expanded as needed
- **Stacks**—Structures in which the last item added is the first item removed
- **Hash maps**—Objects stored and retrieved using unique keys

These data structures are part of the `java.util` package, a collection of useful classes for handling data, dates, strings, and other things. The addition of generics and new `for` loops for iteration enhances their capabilities.

You also were introduced to enumerations, a data type for representing a set of related values as constants.

Learning about the ways in which you can organize data in Java has benefits in all aspects of software development. Whether you're learning the language to write servlets, desktop applications, apps, or something else, you need to represent data in numerous ways.

Q&A

Q The `HolidaySked` project from today could be implemented as an array of Boolean values. Is one way preferable to the other?

A That depends. One thing you'll find as you work with data structures is that there are often many ways to implement something. Bit sets are somewhat preferable to a Boolean array when the size of your program matters, because a bit set is smaller. An array of a primitive type such as Boolean is preferable when the speed of your program matters, because arrays are somewhat faster. In the example of the `HolidaySked` class, it's so small that the difference is negligible, but as you develop your own robust, real-world applications, these kinds of decisions can sometimes make a difference.

Q The Java compiler's warning for data structures that don't use generics is pretty ominous. It doesn't sound like a very good idea to release a class that has "unchecked or unsafe operations." Is there any reason to stick with old code or not use generics with data structures?

A The compiler's warning about safety is a bit overstated. Java programmers have been using array lists, hash maps, and other structures for years in their classes, creating software that runs reliably and safely. The lack of generics meant that more work was necessary to ensure that runtime problems didn't occur because of wrong classes being placed in a structure.

It's more accurate to state that data structures can be made more safe through the use of generics, rather than suggesting that previous versions of Java were unsafe.

My personal rule is to use generics in new code and old code that's being reorganized or significantly rewritten, but leave alone old code that works correctly.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which of the following kinds of data cannot be stored in a hash map?

A. `String`

B. `int`

C. Both can be stored in a map.

2. An array list is created, and three strings, "Tinker", "Evers", and "Chance", are added to it. The method `remove("Evers")` is called. Which of the following `ArrayList` methods retrieves the string "Chance"?

A. `get(1);`

B. `get(2);`

C. `get("Chance");`

3. Which of these classes implements the `Map` interface?

A. `Stack`

B. `HashMap`

C. `BitSet`

Answers

1. C. In past versions of Java, to store primitive types such as `int` in a map, objects had to be used to represent their values (such as `Integer` for integers). This is no longer true. Primitive types are converted automatically to the corresponding object class through a process called autoboxing.

2. A. The index numbers of each item in an array list can change as items are added or removed. Because "Chance" becomes the second item in the list after "Evers" is removed, it is retrieved by calling `get(1)`.

3. B. `HashMap` implements the interface, as does a similar class called `Hashtable`.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class Recursion {
    public int dex = -1;

    public Recursion() {
        dex = getValue(17);
    }

    public int getValue(int dexValue) {
        if (dexValue > 100) {
            return dexValue;
        } else {
            return getValue(dexValue * 2);
        }
    }

    public static void main(String[] arguments) {
        Recursion r = new Recursion();
        System.out.println(r.dex);
    }
}
```

What will be the output of this application?

A. –1

B. 17

C. 34

D. 136

The answer is available on the book's website at www.java21days.com. Visit the Day 8 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Add two more conditions to the ComicBooks application: "pristine mint" for books that should sell at 5 times their base price, and "coverless" for books that should sell at one-tenth of their base price.
2. Rewrite the ComicBooks application so that the set of possible conditions of a comic is an enumeration.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 9. Working with Swing

Computer users today expect the software they use to feature a graphical user interface (GUI) with a variety of widgets such as text boxes, sliders, and scrollbars. The Java Class Library includes a set of packages called Swing that enable Java programs to offer a sophisticated GUI and collect user input with the mouse, keyboard, and other input devices.

Today, you will use Swing to create applications that feature these GUI components:

- **Frames**—Windows that can include a title bar; menu bar; and Maximize, Minimize, and Close buttons
- **Containers**—Interface elements that can hold other components
- **Buttons**—Clickable regions with text or graphics indicating their purpose
- **Labels**—Text or graphics that provide information
- **Text fields and text areas**—Windows that accept keyboard input and allow text to be edited
- **Drop-down lists**—Groups of related items that can be selected from drop-down menus or scrolling windows
- **Check boxes and radio buttons**—Small squares or circles that can be selected or deselected
- **Image icons**—Graphics that can be added to buttons, labels, and other components
- **Scrolling panes**—Panels that hold components too big for a user interface that are accessed in full with a scrollbar

Creating an Application

Swing enables you to create a Java program with an interface that adopts the style of the native operating system, such as Windows or Linux, or a style that's unique to Java. Each of these styles is called a *look and feel* because it describes both the appearance of the interface and how its components function when they are used.

Java offers a distinctive new look and feel called Nimbus that's unique to the language.

Swing components are part of the `javax.swing` package, a standard part of the Java Class Library. To refer to a Swing class using its short name—without referring to the package, in other words—you must make it available with an `import` statement or use a catchall statement such as the following:

```
import javax.swing.*;
```

Two other packages that are used to support GUI programming are `java.awt`, the Abstract Windowing Toolkit (AWT), and `java.awt.event`, event-handling classes that handle user input.

When you use a Swing component, you work with objects of that component's class. You create the component by calling its constructor and then calling methods of the component as needed for proper setup.

All Swing components are subclasses of the abstract class `JComponent`. It includes methods to set a component's size, change the background color, define the font used for any displayed text, and set up *ToolTips*. These are explanatory text that appears when you hover the mouse over the component for a few seconds.

Caution

Swing classes inherit from many of the same superclasses as the Abstract Windowing Toolkit, so it is possible to use Swing and AWT components together in the same interface. However, the two types of components will not be rendered correctly in a container, so it's best to always use Swing components—there's one for every AWT component.

Before components can be displayed in a user interface, they must be added to a *container*, a component that can hold other components. Swing containers are subclasses of `java.awt.Container`. This class includes methods to add and remove components from a container, arrange components using an object called a layout manager, and set up borders around the edges of a container. Containers often can be placed in other containers.

Creating an Interface

The first step in creating a Swing application is to create a class that represents the main GUI. An object of this class serves as a container that holds all the other components to be displayed.

In many projects, the main interface object is a frame (the `JFrame` class). Frames are the window shown whenever you open an application on your computer, regardless of the language it was programmed in. Frames have a title bar; Maximize, Minimize, and Close buttons; and other features.

In a graphical environment such as Windows or Mac OS, users expect to be able to move, resize, and close the windows of programs they run. One way to create a graphical Java application is to make the interface a subclass of `JFrame`, as in the following class declaration:

Click here to view code image

```
public class FeedReader extends JFrame {
    // body of class
}
```

The constructor of the class should handle the following tasks:

- Call a superclass constructor to give the frame a title and handle other setup procedures.
- Set the size of the frame's window, either by specifying the width and height in pixels or by letting Swing choose the right size.
- Decide what to do if a user closes the window.
- Display the frame.

The `JFrame` class has the simple constructors `JFrame()` and `JFrame(`*`String`*`)`. One sets the frame's title bar to the specified text, and the other leaves the title bar empty. You also can set the title by calling the frame's `setTitle(`*`String`*`)` method.

The size of a frame can be established by calling the `setSize(`*`int, int`*`)` method with the width and height as arguments. A frame's size is indicated in pixels, so calling `setSize(650, 550)` creates a frame 650 pixels wide and 550 pixels tall, taking up most of a screen that has 800×600 resolution.

Note

You also can call the method `setSize(Dimension)` to set up a frame's size. `Dimension` is a class in the `java.awt` package that represents the width and height of a user interface component. Calling the `Dimension(int, int)` constructor creates a `Dimension` object representing the width and height specified as arguments.

Another way to set a frame's size is to fill the frame with the components it will contain and then call the frame's `pack()` method. This resizes the frame based on the size of the components inside it. If the frame is bigger than it needs to be, `pack()` shrinks it to the minimum size required to display the components. If the frame is too small (or the size has not been set), `pack()` expands it to the required size.

Frames are invisible when they are created. You can make them visible by calling the frame's `setVisible(boolean)` method with the literal `true` as an argument.

If you want a frame to be displayed when it is created, call one of these methods in the constructor. You also can leave the frame invisible, requiring any class that uses the frame to make it visible by calling `setVisible(true)`. As you probably have surmised, calling `setVisible(false)` makes a frame invisible.

When a frame is displayed, the default behavior is for it to be positioned in the upper-left corner of the computer's desktop.

You can specify a different location by calling the `setBounds(int, int, int, int)` method. The first two arguments to this method are the (x,y) position of the frame's upper-left corner on the desktop. The last two arguments set the frame's width and height.

Another way to set the bounds is with a `Rectangle` object from the `java.awt` package. Create the rectangle with the `Rectangle(int, int, int, int)` constructor. The first two arguments are the (x, y) position of the upper-left corner. The next two are the width and height. Call `setBounds(Rectangle)` to draw the frame at that spot.

The following class represents a 300×100 frame with "Edit Payroll" in the title bar:

Click here to view code image

```
public class Payroll extends javax.swing.JFrame {
    public Payroll() {
        super("Edit Payroll");
        setSize(300, 100);
        setVisible(true);
    }
}
```

Every frame has Maximize, Minimize, and Close buttons on the title bar that the user can control—the same controls present in the interface of other software running on your computer.

The normal behavior when a frame is closed is for the application to keep running. When a frame serves as a program's main GUI, this leaves a user with no way to stop the program.

To change this, you must call a frame's `setDefaultCloseOperation()` method with one of four static variables as an argument:

- `EXIT_ON_CLOSE`—Exits the application when the frame is closed
- `DISPOSE_ON_CLOSE`—Closes the frame, removes the frame object from Java Virtual Machine (JVM) memory, and keeps running the application
- `DO_NOTHING_ON_CLOSE`—Keeps the frame open and continues running
- `HIDE_ON_CLOSE`—Closes the frame and continues running

These variables are in the `JFrame` class because it implements the `WindowConstants` interface. To prevent a user from closing a frame, add the following statement to the frame's constructor method:

Click here to view code image

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

If you are creating a frame to serve as an application's main user interface, the expected behavior is probably `EXIT_ON_CLOSE`, which shuts down the application along with the frame.

As mentioned earlier, you can customize the overall appearance of a user interface in Java by designating a look and feel. The `UIManager` class in the `javax.swing` package manages this aspect of Swing. To set the look and feel, call the class method `setLookAndFeel(`*`String`*`)` with the name of

the look and feel's class as the argument. Here's how to choose the Nimbus look and feel:

Click here to view code image

```
UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
 );
```

This method call should be contained within a `try-catch` block because it might generate five different exceptions. Catching the `Exception` class and ignoring it causes the default look and feel to be used in the unlikely circumstance that Nimbus can't be chosen properly.

Caution

Using `EXIT_ON_CLOSE` shuts down the entire JVM, so it should be used only in the frame for an application's main window. If anything needs to happen after the frame closes, `DISPOSE_ON_CLOSE` or `HIDE_ON_CLOSE` should be used instead.

Developing a Framework

Today's first project is an application that displays a frame containing no other interface components. In NetBeans, create a new Java file with the class name `SimpleFrame` and the package name `com.java21days`; then enter Listing 9.1 as the source code. This simple application displays a frame 300×100 pixels in size and can serve as a framework—pun unavoidable—for any applications you create that use a GUI.

LISTING 9.1 The Full Text of `SimpleFrame.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class SimpleFrame extends JFrame {
 6:     public SimpleFrame() {
 7:         super("Frame Title");
 8:         setSize(300, 100);
 9:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10:         setLookAndFeel();
```

```
11:          setVisible(true);
12:      }
13:
14:      private static void setLookAndFeel() {
15:          try {
16:              UIManager.setLookAndFeel(
17:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAnd
Feel"
18:              );
19:          } catch (Exception exc) {
20:              // ignore error
21:          }
22:      }
23:
24:      public static void main(String[] arguments) {
25:          setLookAndFeel();
26:          SimpleFrame sf = new SimpleFrame();
27:      }
28: }
```

When you compile and run the application, you should see the frame displayed in Figure 9.1.

FIGURE 9.1 Displaying a frame.

The SimpleFrame application isn't much to look at. The GUI contains no components, aside from the standard Minimize, Maximize, and Close (X) buttons on the title bar, as shown in Figure 9.1. You add components later today.

In the application, a `SimpleFrame` object is created in the `main()` method in lines 24–27. If you had not displayed the frame when it was constructed, you could call `sf.setVisible(true)` in the `main()` method to display the frame.

Nimbus is set as the frame's look and feel in lines 16–18.

The work involved in creating the frame's user interface takes place in the `SimpleFrame()` constructor in lines 6–12. Components can be created and added to the frame within this constructor.

Creating a Component

Creating a GUI is a great way to get experience working with objects in Java, because each interface component is represented by its own class.

To use an interface component in Java, you create an object of that component's class. You already have worked with the container class `JFrame`.

One of the simplest components to employ is `JButton`, the class that represents clickable buttons.

In most programs, buttons trigger an action. You can click Install to begin installing software, click a smiley button to begin a new game of Angry Birds, click the Minimize button to prevent your boss from seeing Angry Birds running, and so on.

A Swing button can feature a text label, a graphical icon, or a combination of both.

Constructors you can use for buttons include the following:

- `JButton(`*`String`*`)`—A button labeled with the specified text
- `JButton(`*`Icon`*`)`—A button that displays the specified graphical icon
- `JButton(`*`String`*`, `*`Icon`*`)`—A button with the specified text and graphical icon

The following statements create three buttons with text labels:

Click here to view code image

```
JButton play = new JButton("Play");
JButton stop = new JButton("Stop");
JButton rewind = new JButton("Rewind");
```

Graphical buttons are covered later today.

Adding Components to a Container

Before you can display a user interface component such as a button in a Java program, you must add it to a container and display that container.

To add a component to a container, call the container's `add(`*`Component`*`)` method with the component as the argument (all user interface components in Swing inherit from `java.awt.Component`).

The simplest Swing container is a panel (the `JPanel` class). The following example creates a button and adds it to a panel:

Click here to view code image

```
JButton quit = new JButton("Quit");
JPanel panel = new JPanel();
panel.add(quit);
```

Use the same technique to add components to frames and windows.

The `ButtonFrame` class, shown in Listing 9.2, expands on the application framework created earlier today. A panel is created, three buttons are added to the panel, and then it is added to a frame. Enter the source code of Listing 9.2 into a new Java file called `ButtonFrame` in NetBeans, making sure to put it in the `com.java21days` package.

LISTING 9.2 The Full Text of `ButtonFrame.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class ButtonFrame extends JFrame {
 6:     JButton load = new JButton("Load");
 7:     JButton save = new JButton("Save");
 8:     JButton unsubscribe = new JButton("Unsubscribe");
 9:
10:     public ButtonFrame() {
11:         super("Button Frame");
12:         setSize(340, 170);
13:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:         JPanel pane = new JPanel();
15:         pane.add(load);
16:         pane.add(save);
17:         pane.add(unsubscribe);
18:         add(pane);
19:         setVisible(true);
20:     }
21:
22:     private static void setLookAndFeel() {
23:         try {
24:             UIManager.setLookAndFeel(
25:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAnd
Feel"
26:             );
```

```
27:          } catch (Exception exc) {
28:              System.out.println(exc.getMessage());
29:          }
30:      }
31:
32:      public static void main(String[] arguments) {
33:          setLookAndFeel();
34:          ButtonFrame bf = new ButtonFrame();
35:      }
36: }
```

When you run the application, a small frame opens that contains the three buttons, as shown in Figure 9.2.

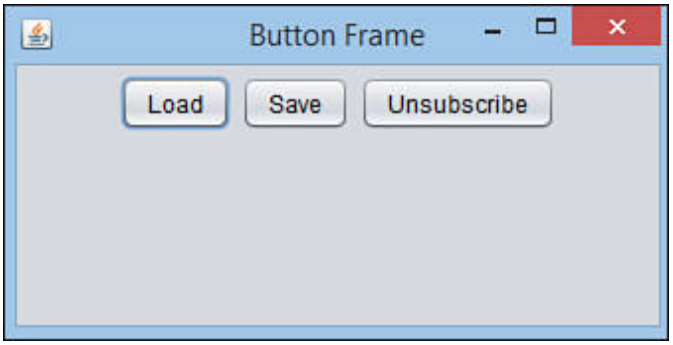


FIGURE 9.2 The ButtonFrame application.

The `ButtonFrame` class has three instance variables: the `load`, `save`, and `unsubscribe` `JButton` objects.

In lines 14–17 of Listing 9.2, a new `JPanel` object is created, and the three buttons are added to the panel by calls to its `add(`*`Component`*`)` method. When the panel contains all the buttons, the frame's own `add(`*`Component`*`)` method is called in line 18 with the panel as an argument, adding it to the frame.

Note

If you click the buttons, nothing happens. Doing something in response to a button click is covered in Day 12, "Responding to User Input."

Working with Components

Swing offers more than two dozen user interface components in addition to the buttons and containers you have used so far. You will work with many of these components for the rest of today and on Day 10, "Building a Swing Interface."

All Swing components share a common superclass, `javax.swing.JComponent`, from which they inherit several methods you will find useful in your own programs.

The `setEnabled(`*`boolean`*`)` method determines whether a component can receive user input (an argument of `true`) or is inactive and cannot receive input (`false`). Components are enabled by default. Many components change in appearance to indicate when they are not presently usable. For instance, a disabled `JButton` has light gray borders and gray text. If you want to check whether a component is enabled, you can call the `isEnabled()` method, which returns a `boolean` value.

The `setVisible(`*`boolean`*`)` method works for all components the way it does for containers. Use `true` to display a component and `false` to hide it. There also is a `boolean isVisible()` method.

The `setSize(`*`int`*`, `*`int`*`)` method resizes the component to the width and height specified as arguments, and `setSize(`*`Dimension`*`)` uses a `Dimension` object to accomplish the same thing. For most components, you don't need to set a size; the default is usually acceptable. To find out a component's size, call its `getSize()` method, which returns a `Dimension` object with the dimensions in `height` and `width` instance variables.

As you will see, similar Swing components also have other methods in common, such as `setText()` and `getText()` for text components and `setValue()` and `getValue()` for components that store a numeric value.

Caution

When you begin working with Swing components, a common source of mistakes is to set up aspects of a component after it has been added to a container. Be sure to set up a component fully before placing it in a panel or any other container.

Image Icons

Swing supports the use of graphical `ImageIcon` objects on buttons and other components in which a label can be provided. An *icon* is a small graphic that can be placed on a button, label, or other user interface element to identify it. Examples include a garbage can or recycling bin icon for deleting files, and folder icons for opening and storing files.

You can create an `ImageIcon` object by specifying the filename of a graphic as the only argument to the constructor. The following example loads an icon from the graphics file `subscribe.gif` and creates a `JButton` with the icon as its label:

Click here to view code image

```
ImageIcon subscribe = new ImageIcon("subscribe.gif");
JButton button = new JButton(subscribe);
JPanel pane = new JPanel();
pane.add(button);
add(pane);
setVisible(true);
```

Listing 9.3 is a Java application that creates four image icons with text labels, adds them to a panel, and then adds the panel to a frame. Create a new empty Java file in NetBeans for a class named `IconFrame` in the package `com.java21days`, and enter this listing with the source code editor.

LISTING 9.3 The Full Text of `IconFrame.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class IconFrame extends JFrame {
 6:     JButton load, save, subscribe, unsubscribe;
 7:
 8:     public IconFrame() {
 9:         super("Icon Frame");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         JPanel panel = new JPanel();
12:         // create icons
13:         ImageIcon loadIcon = new ImageIcon("load.gif");
14:         ImageIcon saveIcon = new ImageIcon("save.gif");
15:         ImageIcon subscribeIcon = new
ImageIcon("subscribe.gif");
```

```
16:          ImageIcon unsubscribeIcon = new
ImageIcon("unsubscribe.gif");
17:          // create buttons
18:          load = new JButton("Load", loadIcon);
19:          save = new JButton("Save", saveIcon);
20:          subscribe = new JButton("Subscribe", subscribeIcon);
21:          unsubscribe = new JButton("Unsubscribe",
unsubscribeIcon);
22:          // add buttons to panel
23:          panel.add(load);
24:          panel.add(save);
25:          panel.add(subscribe);
26:          panel.add(unsubscribe);
27:          // add the panel to a frame
28:          add(panel);
29:          pack();
30:          setVisible(true);
31:      }
32:
33:      public static void main(String[] arguments) {
34:          IconFrame ike = new IconFrame();
35:      }
36: }
```

Figure 9.3 shows the result.

FIGURE 9.3 An interface containing buttons labeled with icons.

The icons' graphics referred to in lines 13–16 can be found on this book's official website at www.java21days.com on the Day 9 page.

In NetBeans, the graphics must be part of the project before this application runs correctly. The graphics need to be stored in the main folder of the Java21 project you've been using throughout this book to hold the classes you create. Follow these steps:

1. Save the graphics files to a temporary folder on your computer.
2. Click the Files tab to bring that pane to the front. The Files tab opens, as shown in Figure 9.4, listing the files in the project.

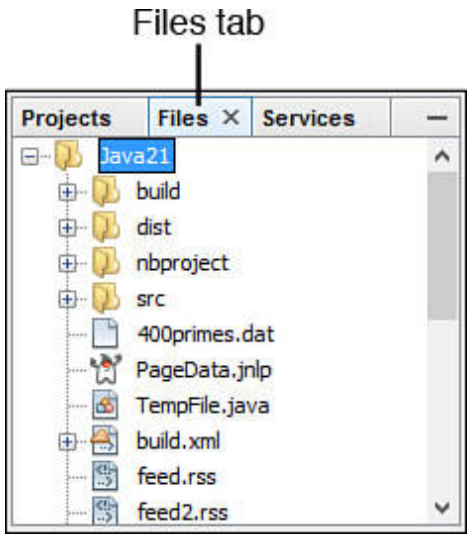


FIGURE 9.4 Dragging files into the NetBeans Files pane.

3. Drag and drop the four graphics files into the `Java21` folder in this pane.

The IconFrame application does not set the size of the frame in pixels. Instead, the `pack()` method is called in line 29 to expand the frame to the minimum size required to present the four buttons next to each other.

If the frame were set to be tall rather than wide—for instance, by calling `setSize(100, 400)` in the constructor—the buttons would be stacked vertically.

Note

Some of the project's graphics are from Oracle's Java Look and Feel Graphics Repository, a collection of icons suitable for use in your own programs. If you're looking for icons to experiment with in Swing applications, you can find some at the following address: www.oracle.com/technetwork/java/index-138612.html.

Labels

A *label* is a user component that holds text, an icon, or both. Labels, which are created from the `JLabel` class, identify the purpose of other components on an interface. A user cannot edit them directly.

To create a label, you can use these simple constructors:

- `JLabel(`*`String`*`)`—A label with the specified text

- `JLabel(String, int)`—A label with the specified text and alignment
- `JLabel(String, Icon, int)`—A label with the specified text, icon, and alignment

A label's alignment determines how its text or icon is aligned in relation to the area taken up by the window. Three static class variables of the `SwingConstants` interface are used to specify alignment: `LEFT`, `CENTER`, and `RIGHT`.

You can set a label's contents with the `setText(String)` or `setIcon(Icon)` methods. You also can retrieve these things with the `getText()` and `getIcon()` methods.

The following statements create three labels with left, center, and right alignment, respectively:

Click here to view code image

```
JLabel feedsLabel = new JLabel("Feeds: ", SwingConstants.LEFT);
JLabel urlLabel = new JLabel("URL: ", SwingConstants.CENTER);
JLabel dateLabel = new JLabel("Date: ", SwingConstants.RIGHT);
```

Text Fields

A *text field* is a location on an interface where a user can enter and modify text using the keyboard. Text fields are represented by the `JTextField` class, and each can handle one line of input. The next section describes a text area component that can handle multiple lines.

Constructors for text fields include the following:

- `JTextField()`—An empty text field
- `JTextField(int)`—A text field with the specified width
- `JTextField(String, int)`—A text field with the specified text and width

A text field's width attribute has relevance only if the interface is organized in a manner that does not resize components. You get more experience with this when you work with layout managers on Day 11, "Arranging Components on a User Interface."

The following statements create an empty text field that has enough space for roughly 60 characters and a text field of the same size with the starting text "Enter feed URL here":

Click here to view code image

```
JTextField rssUrl = new JTextField(60);
JTextField rssUrl2 = new JTextField("Enter feed URL here", 60);
```

Text fields and text areas both inherit from the superclass `JTextComponent` and share many common methods.

The `setEditable(boolean)` method determines whether a text component can be edited (`true`) or not (`false`). An `isEditable()` method returns a corresponding `boolean` value.

The `setText(String)` method changes the text to the specified string, and the `getText()` method returns the component's current text as a string. Another method retrieves only the text that a user has highlighted in the `getSelectedText()` component.

Password fields are text fields that hide the characters a user types into the field. They are represented by the `JPasswordField` class, a subclass of `JTextField`. The `JPasswordField` constructors take the same arguments as those of the parent class.

After you have created a password field, call its `setEchoChar(char)` method to obscure input by replacing each input character with the specified character.

The following statements create a password field and set its echo character to #:

Click here to view code image

```
JPasswordField codePhrase = new JPasswordField(20);
codePhrase.setEchoChar('#');
```

Text Areas

Text areas, editable text fields that can handle more than one line of input, are implemented by the `JTextArea` class, which includes these constructors:

- `JTextArea(int, int)`—A text area with the specified number of rows and columns
- `JTextArea(String, int, int)`—A text area with the specified text, rows, and columns

You can use the `getText()`, `getSelectedText()`, and `setText(String)` methods with text areas as you would text fields. Also, an `append(String)` method adds the specified text at the end of the

current text, and an `insert(String, int)` method inserts the specified text at the indicated position.

The `setLineWrap(boolean)` method determines whether text will wrap to the next line when it reaches the right edge of the component. Call `setLineWrap(true)` to cause line wrapping to occur.

The `setWrapStyleWord(boolean)` method determines what wraps to the next line—either the current word (`true`) or the current character (`false`).

The next project you create, the Authenticator application shown in Listing 9.4, uses several Swing components to collect user input: a text field, a password field, and a text area. Labels also are used to indicate the purpose of each text component. In NetBeans, create an empty Java file called `Authenticator` in the package `com.java21days`.

LISTING 9.4 The Full Text of `Authenticator.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class Authenticator extends javax.swing.JFrame {
 6:     JTextField username = new JTextField(15);
 7:     JPasswordField password = new JPasswordField(15);
 8:     JTextArea comments = new JTextArea(4, 15);
 9:     JButton ok = new JButton("OK");
10:     JButton cancel = new JButton("Cancel");
11:
12:     public Authenticator() {
13:         super("Account Information");
14:         setSize(300, 220);
15:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16:
17:         JPanel pane = new JPanel();
18:         JLabel usernameLabel = new JLabel("Username: ");
19:         JLabel passwordLabel = new JLabel("Password: ");
20:         JLabel commentsLabel = new JLabel("Comments: ");
21:         comments.setLineWrap(true);
22:         comments.setWrapStyleWord(true);
23:         pane.add(usernameLabel);
24:         pane.add(username);
25:         pane.add(passwordLabel);
26:         pane.add(password);
```

```
27:          pane.add(commentsLabel);
28:          pane.add(comments);
29:          pane.add(ok);
30:          pane.add(cancel);
31:          add(pane);
32:          setVisible(true);
33:      }
34:
35:      private static void setLookAndFeel() {
36:          try {
37:              UIManager.setLookAndFeel(
38:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAnd
Feel"
39:              );
40:          } catch (Exception exc) {
41:              System.out.println(exc.getMessage());
42:          }
43:      }
44:
45:      public static void main(String[] arguments) {
46:          Authenticator.setLookAndFeel();
47:          Authenticator auth = new Authenticator();
48:      }
49: }
```

This application sets up components and adds them to a panel in lines 17–30. Figure 9.5 shows the application in use. The password is obscured with asterisk characters (*), which is the default when no other echo character is designated by calling the field's `setEchoChar(`*`char`*`)` method.

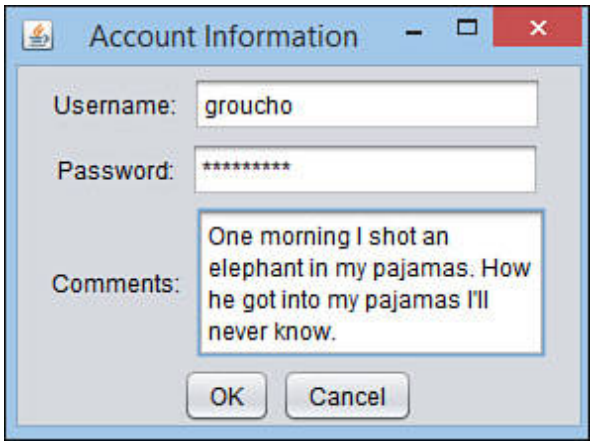


FIGURE 9.5 The Authenticator application.

The text area in this application behaves in a manner that you might not expect. When you reach the bottom of the field and continue entering text, the component grows to make more room for input (and even scrolls below the

bottom edge of the frame). The next section describes how to add scrollbars to prevent the area from changing in size.

Scrolling Panes

Text areas in Swing do not include horizontal or vertical scrollbars, and there's no way to add them using this component alone.

Swing supports scrollbars through a new container that can be used to hold any component that can be scrolled: `JScrollPane`.

A scrolling pane is associated with a component in the pane's constructor. You can use these following constructors:

- `JScrollPane(`*`Component`*`)`—A scrolling pane that contains the specified component
- `JScrollPane(`*`Component`*`, int, int)`—A scrolling pane with the specified component, vertical scrollbar configuration, and horizontal scrollbar configuration

Scrollbars are configured using one of six static class variables of the `ScrollPaneConstants` interface. There are three for vertical scrollbars:

- `VERTICAL_SCROLLBAR_ALWAYS`
- `VERTICAL_SCROLLBAR_AS_NEEDED`
- `VERTICAL_SCROLLBAR_NEVER`

There also are three variables for horizontal scrollbars with similar names.

After you create a scrolling pane containing a component, you should add the pane to containers in place of that component.

The following example creates a text area with a vertical scrollbar and no horizontal scrollbar and then adds it to a container:

Click here to view code image

```
JPanel pane = new JPanel();
JTextArea comments = new JTextArea(4, 15);
JScrollPane scroll = new JScrollPane(comments,
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
pane.add(scroll);
add(pane);
```

Note

This book's website contains Authenticator2, a full application that makes use of this code. Visit www.java21days.com and open the Day 9 page to find a link to `Authenticator2.java`.

Check Boxes and Radio Buttons

The next two components, check boxes and radio buttons, hold only two possible values: selected or not selected.

Check boxes are used to make a simple choice in an interface, such as yes/no or on/off. Radio buttons are grouped so that only one button can be selected at any time.

Check boxes (the `JCheckBox` class) appear as labeled or unlabeled boxes that contain a check mark when they are selected and nothing otherwise. Radio buttons (the `JRadioButton` class) appear as circles that contain a dot when selected and nothing otherwise.

Both the `JCheckBox` and `JRadioButton` classes have several useful methods inherited from `JToggleButton`, their common superclass:

- `setSelected(boolean)`—Selects the component if the argument is `true` and deselects it otherwise
- `isSelected()`—Returns a `boolean` indicating whether the component is currently selected

The following constructors can be used for the `JCheckBox` class:

- `JCheckBox(String)`—A check box with the specified text label
- `JCheckBox(String, boolean)`—A check box with the specified text label that is selected if the second argument is `true`
- `JCheckBox(Icon)`—A check box with the specified graphical icon
- `JCheckBox(Icon, boolean)`—A check box with the specified graphical icon that is selected if the second argument is `true`
- `JCheckBox(String, Icon)`—A check box with the specified text label and graphical icon
- `JCheckBox(String, Icon, boolean)`—A check box with the specified text label and graphical icon that is selected if the third argument is `true`

The `JRadioButton` class has constructors with the same arguments and functionality.

Check boxes and radio buttons by themselves are *nonexclusive*, meaning that if you have five check boxes in a container, all five can be checked or unchecked at the same time. To make them exclusive, as radio buttons should be, you must organize related components into groups.

To organize several radio buttons into a group, allowing only one to be selected at a time, create a `ButtonGroup` class object, as demonstrated in the following statement:

Click here to view code image

```
ButtonGroup choice = new ButtonGroup();
```

The `ButtonGroup` object keeps track of all radio buttons in its group. Call the group's `add(`*`Component`*`)` method to add the specified component to the group.

The following example creates a group and two radio buttons that belong to it:

Click here to view code image

```
ButtonGroup saveFormat = new ButtonGroup();
JRadioButton s1 = new JRadioButton("JSON", false);
saveFormat.add(s1);
JRadioButton s2 = new JRadioButton("XML", true);
saveFormat.add(s2);
```

The `saveFormat` object groups the `s1` and `s2` radio buttons. The `s2` object, which has the label `"XML"`, is selected. Only one member of the group can be selected at a time. If one component is selected, the `ButtonGroup` object ensures that all others in the group are deselected.

Create a new empty Java file in NetBeans called `FormatFrame` in the package `com.java21days`. Enter the source code shown in Listing 9.5 to create an application with four radio buttons in a group.

LISTING 9.5 The Full Text of `FormatFrame.java`

Click here to view code image

```
1: package com.java21days;
2:
3: import javax.swing.*;
4:
```

```
 5: public class FormatFrame extends JFrame {
 6:     JRadioButton[] teams = new JRadioButton[4];
 7:
 8:     public FormatFrame() {
 9:         super("Choose an Output Format");
10:         setSize(320, 120);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         teams[0] = new JRadioButton("Atom");
13:         teams[1] = new JRadioButton("RSS 0.92");
14:         teams[2] = new JRadioButton("RSS 1.0");
15:         teams[3] = new JRadioButton("RSS 2.0", true);
16:         JPanel panel = new JPanel();
17:         JLabel chooseLabel = new JLabel(
18:             "Choose an output format for syndicated news
items.");
19:         panel.add(chooseLabel);
20:         ButtonGroup group = new ButtonGroup();
21:         for (JRadioButton team : teams) {
22:             group.add(team);
23:             panel.add(team);
24:         }
25:         add(panel);
26:         setVisible(true);
27:     }
28:
29:     private static void setLookAndFeel() {
30:         try {
31:             UIManager.setLookAndFeel(
32:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAnd
Feel"
33:             );
34:         } catch (Exception exc) {
35:             System.out.println(exc.getMessage());
36:         }
37:     }
38:
39:     public static void main(String[] arguments) {
40:         FormatFrame.setLookAndFeel();
41:         FormatFrame ff = new FormatFrame();
42:     }
43: }
```

Figure 9.6 shows the application running. The four `JRadioButton` objects are stored in an array in lines 12–15. In the `for` loop in lines 21–24, each element is first added to a button group and then is added to a panel. After the loop ends, the panel is added to the frame.

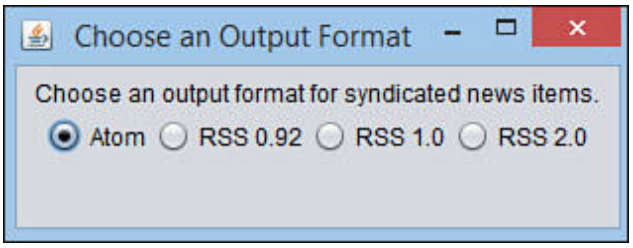


FIGURE 9.6 The FormatFrame application.

Choosing one of the radio buttons causes the existing choice to be deselected.

Combo Boxes

The Swing class `JComboBox` can be used to create combo boxes, components that present a drop-down menu from which a single value can be selected. The menu is hidden when the component is not being used, thus taking up less space in a GUI.

After a combo box is created by calling the `JComboBox()` constructor with no arguments, the combo box's `addItem(`*`Object`*`)` method adds items to the list.

Another way to create a combo box is to call `JComboBox(`*`Object[]`*`)` with an array that contains the items. If the items are text, a `String` array would be the argument.

In a combo box, users can select only one of the items on the drop-down menu. If the component's `setEditable()` method is called with `true` as an argument, it also supports text entry. This feature gives combo boxes their name: A component configured in this manner serves as both a drop-down menu and a text field.

The `JComboBox` class has several methods you can use to control a drop-down list or combo box:

- `getItemAt(`*`int`*`)`—Returns the text of the list item at the index position specified by the integer argument. As with arrays, the first item of a choice list is at index position 0, the second is at position 1, and so on.
- `getItemCount()`—Returns the number of items in the list.
- `getSelectedIndex()`—Returns the index position of the currently selected item in the list.

- `getSelectedItem()`—Returns the text of the currently selected item.
- `setSelectedIndex(int)`—Selects the item at the indicated index position.
- `setSelectedIndex(Object)`—Selects the specified object in the list.

The FormatFrame2 application, shown in Listing 9.6, rewrites the preceding radio button example. The program uses a noneditable combo box from which a user can choose one of four options.

LISTING 9.6 The Full Text of `FormatFrame2.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class FormatFrame2 extends JFrame {
 6:     String[] formats = { "Atom", "RSS 0.92", "RSS 1.0", "RSS
2.0" };
 7:     JComboBox formatBox = new JComboBox(formats);
 8:
 9:     public FormatFrame2() {
10:         super("Choose a Format");
11:         setSize(220, 150);
12:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13:         JPanel pane = new JPanel();
14:         JLabel formatLabel = new JLabel("Output formats:");
15:         pane.add(formatLabel);
16:         pane.add(formatBox);
17:         add(pane);
18:         setVisible(true);
19:     }
20:
21:     private static void setLookAndFeel() {
22:         try {
23:             UIManager.setLookAndFeel(
24:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAnd
Feel"
25:             );
26:         } catch (Exception exc) {
27:             System.out.println(exc.getMessage());
28:         }
29:     }
```

```
30:
31:     public static void main(String[] arguments) {
32:         FormatFrame2.setLookAndFeel();
33:         FormatFrame2 ff = new FormatFrame2();
34:     }
35: }
```

A string array is defined in line 6, and then these strings are used in the combo box constructor in line 7 to set its possible values. Figure 9.7 shows the application as the combo box is expanded so that a value can be selected.

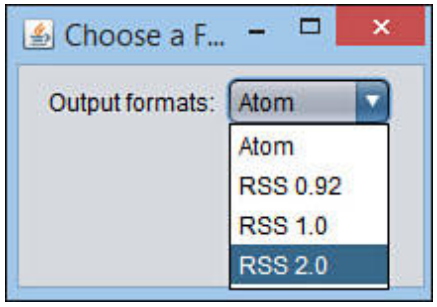


FIGURE 9.7 The FormatFrame2 application.

Lists

The last Swing component to be introduced today is similar to combo boxes. Lists, which are represented by the `JList` class, allow you to select one or more values from a list.

You can create and fill lists with the contents of an array or vector (a data structure similar to array lists). The following constructors are available:

- `JList()`—Creates an empty list
- `JList(Object[])`—Creates a list that contains an array of the specified class (such as `String`)
- `JList(Vector<Class>)`—Creates a list that contains the specified `java.util.Vector` object of the specified class

An empty list can be filled by calling its `setListData()` method with either an array or vector as the only argument.

Unlike combo boxes, lists display more than one of their rows when they are presented in a user interface. The default is to display eight items. To change this, call `setVisibleRowCount(int)` with the number of items to display.

The `getSelectedValuesList()` method returns a list of objects containing all the items selected in the list. This list can be cast to an `ArrayList`.

You can use generics with `JList` to indicate the class of the object array the list contains.

The Subscriptions application in the `com.java21days` package, shown in Listing 9.7, displays eight items from an array of strings.

LISTING 9.7 The Full Text of `Subscriptions.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class Subscriptions extends JFrame {
 6:     String[] subs = { "Burningbird", "Freeform Goodness",
 7:         "Ideoplex", "Inessential", "Intertwingly", "Now
This",
 8:         "Rasterweb", "RC3", "Whole Lotta Nothing",
"Workbench" };
 9:     JList<String> subList = new JList<>(subs);
10:
11:     public Subscriptions() {
12:         super("Subscriptions");
13:         setSize(150, 335);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         JPanel panel = new JPanel();
16:         JLabel subLabel = new JLabel("RSS Subscriptions:");
17:         panel.add(subLabel);
18:         subList.setVisibleRowCount(8);
19:         JScrollPane scroller = new JScrollPane(subList);
20:         panel.add(scroller);
21:         add(panel);
22:         setVisible(true);
23:     }
24:
25:     private static void setLookAndFeel() {
26:         try {
27:             UIManager.setLookAndFeel(
28:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAnd
Feel"
29:             );
30:         } catch (Exception exc) {
31:             System.out.println(exc.getMessage());
```

```
32:             }
33:         }
34:
35:         public static void main(String[] arguments) {
36:             Subscriptions.setLookAndFeel();
37:             Subscriptions app = new Subscriptions();
38:         }
39: }
```

The application is shown in Figure 9.8. The Subscriptions application has an interface with a label atop a list displaying eight items. A Scroll pane is used in lines 19–21 to enable the list to be scrolled to see items 9 and 10.

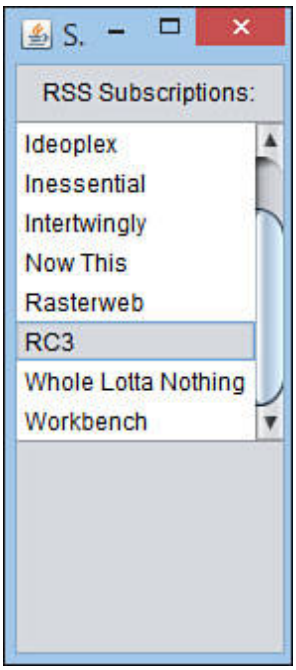


FIGURE 9.8 The Subscriptions application.

The Java Class Library

The first week of this book was devoted to the building blocks of the Java language, including statements, expressions, and operators; and the components of object-oriented programming (OOP) such as methods, constructors, classes, and interfaces.

The second week covers how to build things with those blocks by using the Java Class Library. A lot of your work as a programmer is done for you, provided you know where to look.

The Java Class Library contains over 4,200 classes. Many of them will be useful in programs that you create.

Note

There also are Java class libraries produced by other organizations. The Apache Project has more than a dozen Java open source projects, including HttpComponents, a set of classes for creating web servers, clients, and crawlers. To see these projects, visit http://projects.apache.org.

Oracle offers comprehensive documentation for the library on the Web. A page from this documentation is shown in Figure 9.9.

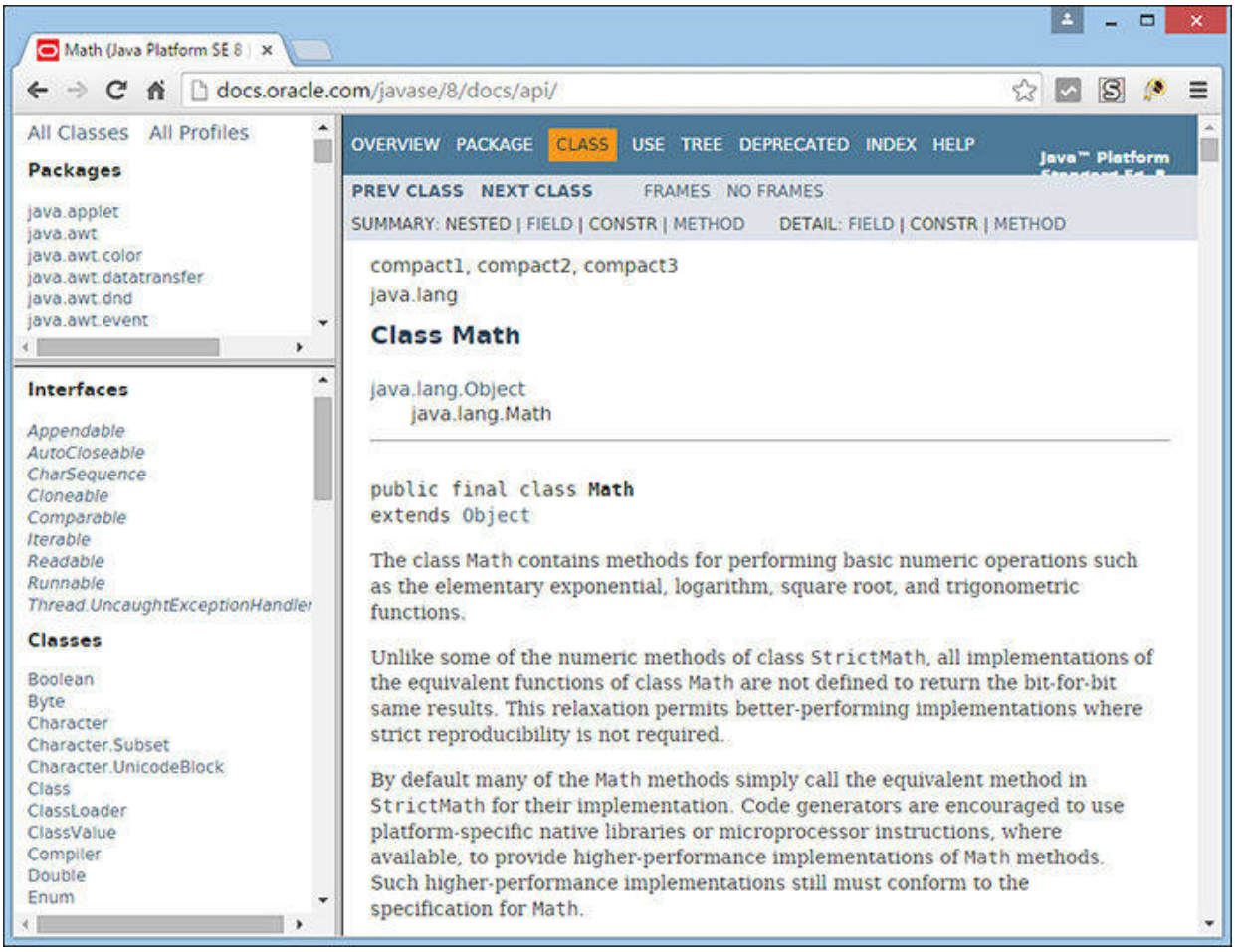


FIGURE 9.9 The Java Class Library's online documentation.

The home page is divided into frames. The largest lists all the packages that compose the library with a description of each. Package names describe their purpose, such as the `java.io` package of classes for input and output from files, Internet servers, and other data sources; and `java.time` for time and date classes.

On the home page, the largest frame presents a list of packages with a short description of each one. Click the name of a package to load a page listing all of its classes.

Each class in the library has its own page of documentation. To get a taste of how to use this reference, follow these steps:

1. In your web browser, load the page http://docs.oracle.com/javase/8/docs/api.
2. Scroll down to the `java.lang` package and click the link. That package's page opens.
3. Scroll down to the link for the Math class, and click it. The page for the class opens.
4. Find the `random()` method link and click it. The page jumps to that section.

The `Math` class page describes its purpose and package. Use a class page to learn how to create an object of the class and what variables and methods it contains.

This class has handy methods that extend Java's math capabilities and turn up often in Java applications. One is `random()`, a method that produces a random `double` value from 0.0 to 1.0.

Here's a statement that uses this method:

Click here to view code image

```
double d100 = Math.random() * 100;
```

The `random()` method produces a randomly generated number ranging from 0.0 up to 1.0, but not including that maximum value. This is a floating-point number, so it needs to be stored in a `float` or `double`.

Because this random number is multiplied by 100, the number will be anything from 0 to 100 (not including 100).

Here's a statement to round the number down to the nearest integer and add 1:

```
d100 = Math.floor(roll) + 1;
```

This statement uses another method of the `Math` class, `floor()`, which rounds a floating-point number down to the closest lower integer. A value of 47.52 would be rounded down to 47. Adding one makes the value of `d100` 48.

Without the `Math` class, you'd have to create your own class to produce random numbers, which is a highly complex task.

Poking around the Java Class Library documentation is a good way to find classes that will save you an enormous amount of time.

Because you're new to Java, you likely will find some of the documentation difficult to understand—it's written for experienced programmers. But as you read this book and encounter interesting Java classes, use this reference to find out more about them. A good place to begin is to look up the methods in a class, each of which performs a job, and see what arguments they take and values they return.

While you are learning about Swing user interface components and classes during the next five days, check out their pages in the official documentation. They have more cool methods than this book has time to cover.

Summary

Today you began working with Swing, the package of classes that enables your Java programs to support a GUI.

You used more than a dozen classes today, creating interface components such as buttons, labels, and text fields. You put each of these into containers: components that include panels, frames, and windows.

This kind of programming can be complicated. Swing represents the largest package of classes that a new Java programmer must deal with in learning the language.

However, as you have experienced with components such as text areas and text fields, Swing components have many superclasses in common. This makes it easier to extend your knowledge into new components and containers, along with the other aspects of Swing programming you will explore over the coming days.

Q&A

Q Is there a way to change the font of text that appears on a button and other components?

A The `JComponent` class includes a `setFont(`*`Font`*`)` method that can be used to set the font for text displayed by that component. You will

work with `Font` objects, color, and more graphics on Day 13, "Creating Java2D Graphics."

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which of the following user interface components is not a container?
   - **A.** `JScrollPane`
   - **B.** `JTextArea`
   - **C.** `JPanel`
2. Which component can be placed into a Scroll pane?
   - **A.** `JTextArea`
   - **B.** `JTextField`
   - **C.** Any component
3. If you use `setSize()` on an application's main frame, where will it appear on your desktop?
   - **A.** At the center of the desktop
   - **B.** At the same spot the last application appeared
   - **C.** At the upper-left corner of the desktop

Answers

1. B. A `JTextArea` requires a container to support scrolling, but it is not a container itself.
2. C. Any component can be added to a Scroll pane, but most are unlikely to need scrolling.
3. C. This is a trick question. Calling `setSize()` has nothing to do with a window's position on the desktop. You must call `setBounds()` rather than `setSize()` to choose where a frame will appear.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
import javax.swing.*;

public class Display extends JFrame {
    public Display() {
        super("Display");
        // answer goes here
        JLabel hello = new JLabel("Hello");
        JPanel pane = new JPanel();
        add(hello);
        pack();
        setVisible(true);
    }

    public static void main(String[] arguments) {
        Display ds = new Display();
    }
}
```

What statement needs to replace `// answer goes here` to make the application function properly?

A. `setSize(300, 200);`

B. `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`

C. `Display ds = new Display();`

D. No statement is needed.

The answer is available on the book's website at www.java21days.com. Visit the Day 9 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create an application with a frame that includes several DVR controls as individual components: play, stop/eject, rewind, fast-forward, and

pause. Choose a size for the window that enables all the components to be displayed on a single row.

2. Create a frame that opens a smaller frame with fields asking for a username and password.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 10. Building a Swing Interface

Although computers can be operated in a command-line environment such as a Linux shell or the Windows command prompt, most computer users expect software to feature a graphical user interface (GUI) and to receive input with a mouse and keyboard.

GUI software can be one of the more challenging tasks for a novice programmer, but as you learned yesterday, Java has simplified the process with Swing.

Swing offers the following features:

- Common user interface components, including buttons, text fields, text areas, labels, check boxes, radio buttons, scrollbars, lists, menu items, and sliders.
- Containers—interface components that can be used to hold other components (including other containers). Containers include frames, panels, menus, menu bars, and tabbed panes.

Swing Features

Most components and containers you learned about yesterday were Swing versions of classes that were part of the Abstract Windowing Toolkit, the original Java package for GUI programming.

Swing offers many additional new components, including keyboard mnemonics, ToolTips, and standard dialog boxes.

Standard Dialog Boxes

The `JOptionPane` class offers several methods you can use to create standard dialog boxes: small windows that ask a question, warn a user, or provide an important message. Figure 10.1 shows an example.

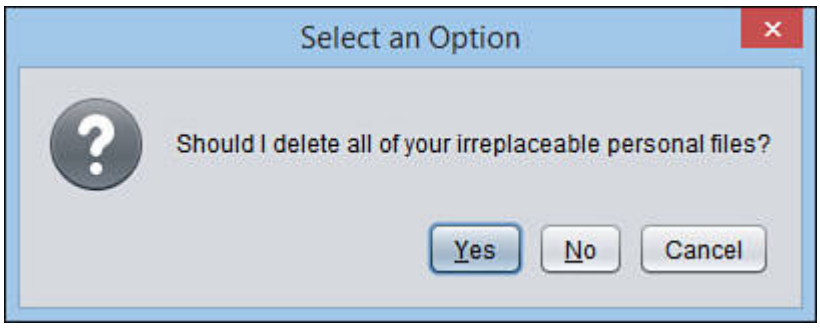


FIGURE 10.1 A standard dialog box.

You have doubtless seen dialog boxes like the one shown in Figure 10.1. When your system crashes, a dialog box appears to break the bad news. When you delete files, a dialog box pops up to make sure that you really want to do so.

These windows are an effective way to communicate with a user without the overhead of creating a new class to represent the window, adding components to it, and writing event-handling methods to receive input. All these tasks are handled automatically when one of the standard dialog boxes offered by `JOptionPane` is used.

The four classes of the standard dialog boxes are as follows:

- `ConfirmDialog`—Asks a question, with buttons for Yes, No, and Cancel responses
- `InputDialog`—Prompts for text input
- `MessageDialog`—Displays a message
- `OptionDialog`—Comprises all three of the other dialog box types

Each of these dialog boxes has its own display method in the `JOptionPane` class.

Confirm Dialog Boxes

The easiest way to create a Yes/No/Cancel dialog box is by calling the `showConfirmDialog (`*`Component`*`, `*`Object`*`)` method. The *`Component`* argument specifies the container that's the parent of the dialog box, which determines where the dialog window should be displayed. If `null` is used instead of a container, or if the container is not a `JFrame` object, the dialog box will be centered onscreen.

The second argument, *`Object`*, can be a string, a component, or an `Icon` object. If it's a string, that text will be displayed in the dialog box. If it's a

component or an `Icon`, that object will be displayed in place of a text message.

This method returns one of five possible integer values, each a class constant of `JOptionPane`: `YES_OPTION`, `NO_OPTION`, `CANCEL_OPTION`, `OK_OPTION`, or `CLOSED_OPTION`.

The following example uses a confirm dialog box with a text message and stores the response in the `response` variable:

Click here to view code image

```
int response = JOptionPane.showConfirmDialog(null,
    "Should I delete all of your irreplaceable personal
files?");
```

This dialog box was shown in Figure 10.1.

Another method offers more options for the dialog box: `showConfirmDialog (`*`Component`*`, `*`Object`*`, `*`String`*`, `*`int`*`, `*`int`*`)`. The first two arguments are the same as those in other `showConfirmDialog()` methods. The last three arguments are the following:

- A string that will be displayed in the dialog box's title bar.
- An integer that indicates which option buttons will be shown. It should be equal to one of the class constants `YES_NO_CANCEL_OPTION` or `YES_NO_OPTION`.
- An integer that describes the kind of dialog box it is, using the class constants `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, or `WARNING_MESSAGE`. (This argument is used to determine which icon to draw in the dialog box along with the message.)

For example:

Click here to view code image

```
int response = JOptionPane.showConfirmDialog(null,
    "Error reading file. Want to try again?",
    "File Input Error",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.ERROR_MESSAGE);
```

Figure 10.2 shows the resulting dialog box.

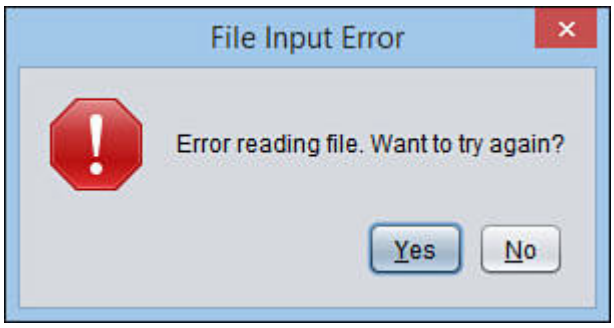


FIGURE 10.2 A confirm dialog box with Yes and No buttons.

Input Dialog Boxes

An input dialog box asks a question and uses a text field to store the response. Figure 10.3 shows an example.

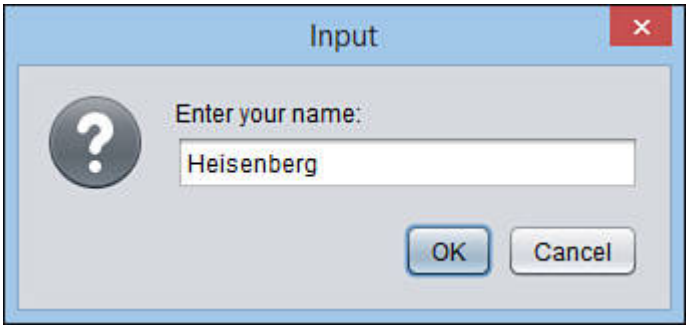


FIGURE 10.3 An input dialog box.

The easiest way to create an input dialog box is with a call to the `showInputDialog(`*`Component`*`,` *`Object`*`)` method. The arguments are the parent component and the string, component, or icon to display in the box.

The input dialog box method call returns a string that represents the user's response. The following statement creates the input dialog box shown in Figure 10.3:

Click here to view code image

```
String response = JOptionPane.showInputDialog(null,
    "Enter your name:");
```

You also can create an input dialog box with the `showInputDialog (`*`Component`*`,` *`Object`*`,` *`String`*`,` *`int`*`)` method. The first two arguments are the same as the shorter method call, and the last two are the following:

- The title to display in the dialog box title bar

- One of five class constants describing the type of dialog box: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, or `WARNING_MESSAGE`

The following statement uses this method to create an input dialog box:

Click here to view code image

```
String response = JOptionPane.showInputDialog(null,
    "What is your ZIP code?",
    "Enter ZIP Code",
    JOptionPane.QUESTION_MESSAGE);
```

Message Dialog Boxes

A message dialog box is a simple window that displays information, as shown in Figure 10.4.

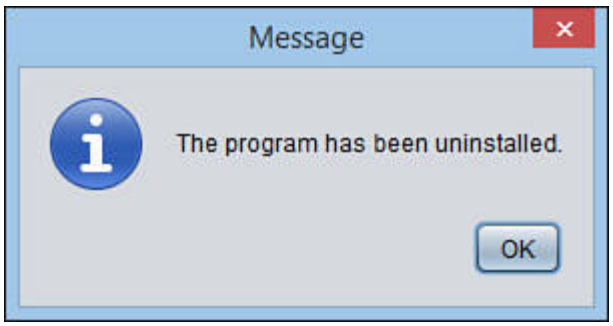


FIGURE 10.4 A message dialog box.

A message dialog box can be created with a call to the `showMessageDialog(`*`Component`*`,` *`Object`*`)` method. As with other dialog boxes, the arguments are the parent component and the string, component, or icon to display.

Unlike the other dialog boxes, message dialog boxes do not return a response value. The following statement creates the message dialog box shown in Figure 10.4:

Click here to view code image

```
JOptionPane.showMessageDialog(null,
    "The program has been uninstalled.");
```

You also can create a message input dialog box by calling the `showMessageDialog(`*`Component`*`,` *`Object`*`,` *`String`*`,` *`int`*`)` method. The use is identical to the `showInputDialog()` method, with

the same arguments, except that `showMessageDialog()` does not return a value.

The following statement creates a message dialog box using this method:

Click here to view code image

```
JOptionPane.showMessageDialog(null,
    "An asteroid has destroyed the Earth.",
    "Asteroid Destruction Alert",
    JOptionPane.WARNING_MESSAGE);
```

Option Dialog Boxes

The most complex of the dialog boxes is the option dialog box, which combines the features of all the other dialog boxes. It can be created with the `showOptionDialog(`*`Component`*`, `*`Object`*`, `*`String`*`, `*`int`*`, `*`int`*`, `*`Icon`*`, `*`Object[]`*`, `*`Object`*`)` method.

The arguments to this method are as follows:

- The parent component of the dialog box
- The text, icon, or component to display
- A string to display in the title bar
- The type of box, using the class constant `YES_NO_OPTION` or `YES_NO_CANCEL_OPTION`, or the value 0 if other buttons will be used instead
- The icon to display, using the class constants `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, or `WARNING_MESSAGE`, or the value 0 if none of these should be used
- An `Icon` object to display instead of one of the icons in the preceding argument
- An array of objects holding the objects that represent the choices in the dialog box if `YES_NO_OPTION` and `YES_NO_CANCEL_OPTION` are not being used
- The object representing the default selection if `YES_NO_OPTION` and `YES_NO_CANCEL_OPTION` are not being used

The final two arguments offer a wide range of possibilities for the dialog box. You can create an array of strings that holds the text of each button to display on the dialog box.

The following example creates an option dialog box that uses an array of `String` objects for the options in the box and the `gender[2]` element as the default selection:

Click here to view code image

```
String[] gender = {
    "Male",
    "Female",
    "None of Your Business"
};
int response = JOptionPane.showOptionDialog(null,
    "What is your gender?",
    "Gender",
    0,
    JOptionPane.INFORMATION_MESSAGE,
    null,
    gender,
    gender[2]);
System.out.println("You chose " + gender[response]);
```

Figure 10.5 shows the resulting dialog box.

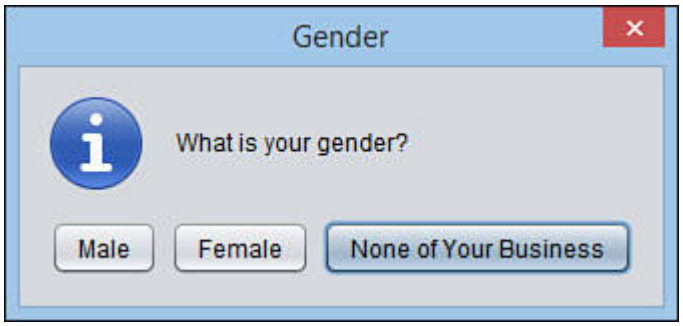


FIGURE 10.5 An option dialog box.

Using Dialog Boxes

The next project shows a series of dialog boxes in a working program. The FeedInfo application in the `com.java21days` package uses dialog boxes to get information from the user; that information is then placed into text fields in the application's main window.

Enter Listing 10.1 and save the result.

LISTING 10.1 The Full Text of `FeedInfo.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.GridLayout;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class FeedInfo extends JFrame {
 8:     private JLabel nameLabel = new JLabel("Name: ",
 9:         SwingConstants.RIGHT);
10:     private JTextField name;
11:     private JLabel urlLabel = new JLabel("URL: ",
12:         SwingConstants.RIGHT);
13:     private JTextField url;
14:     private JLabel typeLabel = new JLabel("Type: ",
15:         SwingConstants.RIGHT);
16:     private JTextField type;
17:
18:     public FeedInfo() {
19:         super("Feed Information");
20:         setSize(400, 145);
21:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22:         setLookAndFeel();
23:         // Site name
24:         String response1 = JOptionPane.showInputDialog(null,
25:             "Enter the site name:");
26:         name = new JTextField(response1, 20);
27:
28:         // Site address
29:         String response2 = JOptionPane.showInputDialog(null,
30:             "Enter the site address:");
31:         url = new JTextField(response2, 20);
32:
33:         // Site type
34:         String[] choices = { "Personal", "Commercial",
"Unknown" };
35:         int response3 = JOptionPane.showOptionDialog(null,
36:             "What type of site is it?",
37:             "Site Type",
38:             0,
39:             JOptionPane.QUESTION_MESSAGE,
40:             null,
41:             choices,
42:             choices[0]);
43:         type = new JTextField(choices[response3], 20);
44:
45:         setLayout(new GridLayout(3, 2));
46:         add(nameLabel);
47:         add(name);
48:         add(urlLabel);
49:         add(url);
```

```
50:             add(typeLabel);
51:             add(type);
52:             setLookAndFeel();
53:             setVisible(true);
54:         }
55:
56:         private void setLookAndFeel() {
57:             try {
58:                 UIManager.setLookAndFeel(
59:                     "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
60:                 );
61:                 SwingUtilities.updateComponentTreeUI(this);
62:             } catch (Exception e) {
63:                 System.err.println("Couldn't use the system "
64:                     + "look and feel: " + e);
65:             }
66:         }
67:
68:         public static void main(String[] arguments) {
69:             FeedInfo frame = new FeedInfo();
70:         }
71: }
```

After you fill in the fields in each dialog box, you see the application's main window, which is displayed in Figure 10.6. Three text fields have values supplied by dialog boxes.

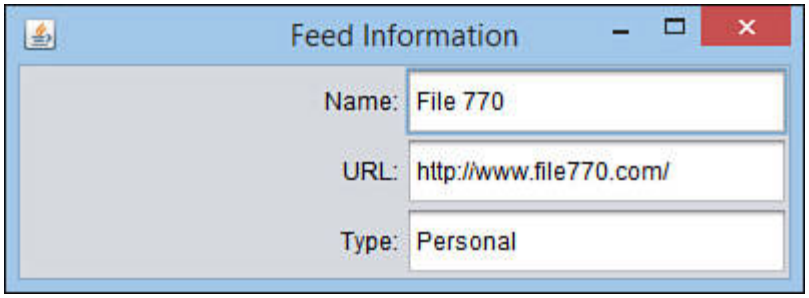


FIGURE 10.6 The main window of the FeedInfo application.

Much of this application is boilerplate code that can be used with any Swing application. The following lines relate to the dialog boxes:

- In lines 24–26, an input dialog box asks the user to enter a site name. This name is used in the constructor for a `JTextField` object, placing it in the text field.
- In lines 29–31, a similar input dialog box asks for a site address, which is used in the constructor for another `JTextField` object.

- In line 34, an array of `String` objects called `choices` is created, and three elements are given values.
- In lines 35–42, an option dialog box asks for the site type. The `choices` array is the seventh argument, which sets up three buttons on the dialog box labeled with the strings in the array: `"Personal"`, `"Commercial"`, and `"Unknown"`. The last argument, `choices[0]`, designates the first array element as the default selection in the dialog box.
- Line 43 contains the response to the option dialog box—an integer identifying the array element that was selected. It is stored in a `JTextField` component called `type`.

The look and feel, which is established in the `setLookAndFeel()` method in lines 56–66, is called at the beginning and end of the frame's constructor method. Because you're opening several dialog boxes in the constructor, you must set up the look and feel before opening them.

This class designates a look and feel differently than previous examples today and on Day 9, "Working with Swing." The `setLookAndFeel()` method is called within the constructor in line 22. To ensure that all components in the user interface reflect the look and feel, the `SwingUtilities` class method `SwingUtilities.updateComponentTreeUI (`*`Component`*`)` is called with `this` as the argument, which refers to the `FeedInfo` object being created.

Sliders

Sliders, which are implemented in Swing with the `JSlider` class, enable the user to set a number by sliding a control within the range of a minimum and maximum value. In many cases, a slider can be used for numeric input instead of a text field. This has the advantage of restricting input to a range of acceptable values.

Figure 10.7 shows a `JSlider` component.

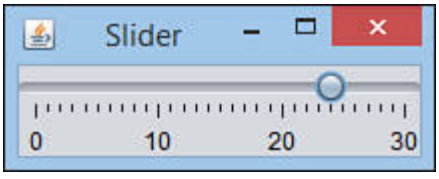


FIGURE 10.7 A `JSlider` component.

Sliders are horizontal by default. You can explicitly set the orientation using two class constants of the `SwingConstants` interface: `HORIZONTAL` or `VERTICAL`.

You can use the following constructor methods:

- `JSlider(int)`—A slider with the specified orientation, a minimum value of 0, maximum value of 100, and starting value of 50
- `JSlider(int, int)`—A slider with the specified minimum value and maximum value
- `JSlider(int, int, int)`—A slider with the specified minimum value, maximum value, and starting value
- `JSlider(int, int, int, int)`—A slider with the specified orientation, minimum value, maximum value, and starting value

Slider components have an optional label that can be used to indicate the minimum value, maximum value, and two different sets of tick marks ranging between the values. The default values are a minimum of 0, maximum of 100, starting value of 50, and horizontal orientation.

The elements of this label are established by calling several methods of `JSlider`:

- `setMajorTickSpacing(int)`—Separates major tick marks by the specified distance. The distance is not in pixels, but in values between the minimum and maximum values represented by the slider.
- `setMinorTickSpacing(int)`—Separates minor tick marks by the specified distance. Minor ticks are displayed as half the height of major ticks.
- `setPaintTicks(boolean)`—Determines whether the tick marks should be displayed (`true`) or not (`false`).
- `setPaintLabels(boolean)`—Determines whether the slider's numeric label should be displayed (`true`) or not (`false`).

These methods should be called on the slider before it is added to a container.

Listing 10.2 contains the `Slider.java` source code; the application was shown in Figure 10.7.

LISTING 10.2 The Full Text of `Slider.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class Slider extends JFrame {
 6:
 7:     public Slider() {
 8:         super("Slider");
 9:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10:         setLookAndFeel();
11:         JSlider pick = new JSlider(JSlider.HORIZONTAL, 0,
30, 5);
12:         pick.setMajorTickSpacing(10);
13:         pick.setMinorTickSpacing(1);
14:         pick.setPaintTicks(true);
15:         pick.setPaintLabels(true);
16:         add(pick);
17:         pack();
18:         setVisible(true);
19:     }
20:
21:     private void setLookAndFeel() {
22:         try {
23:             UIManager.setLookAndFeel(
24:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
25:             );
26:             SwingUtilities.updateComponentTreeUI(this);
27:         } catch (Exception e) {
28:             System.err.println("Couldn't use the system "
29:                 + "look and feel: " + e);
30:         }
31:     }
32:
33:     public static void main(String[] arguments) {
34:         Slider frame = new Slider();
35:     }
36: }
```

Lines 12–16 contain the code that's used to create a `JSlider` component, set up its tick marks to be displayed, and add the component to a container. The rest of the program is a basic framework for an application that consists of a main `JFrame` container with no menus.

Scroll Panes

In early versions of Java, text areas and some other components had a built-in scrollbar. The bar could be used when the text in the component took up more space than the component could display. Scrollbars could be used in either the vertical or horizontal direction to scroll through the text.

One of the most common examples of scrolling is in a web browser, where a scrollbar can be used on any page bigger than the browser's display area.

Swing changes the rules for scrollbars to the following:

- For a component to be able to scroll, it must be added to a `JScrollPane` container.
- This `JScrollPane` container is added to a container in place of the scrollable component.

Scroll panes can be created using the `JScrollPane(`*`Object`*`)` constructor, where *`Object`* represents the component that can be scrolled.

The following example creates a text area in a Scroll pane called `scroller` and then adds it to a container called `mainPane`:

Click here to view code image

```
JTextArea textBox = new JTextArea(7, 30);
JScrollPane scroller = new JScrollPane(textBox);
mainPane.add(scroller);
```

As you work with a Scroll pane, it often can be useful to indicate the size you want it to occupy on the interface. You do so by calling the `setPreferredSize(`*`Dimension`*`)` method of the Scroll pane before adding it to a container. The `Dimension` object represents the width and height of the preferred size in pixels.

The following code builds on the previous example by setting the preferred size of `scroller`:

Click here to view code image

```
Dimension pref = new Dimension(350, 100);
scroller.setPreferredSize(pref);
```

You should set the dimensions before `scroller` is added to a container.

Caution

This is one of many situations in Swing where you must do something in the proper order for it to work correctly. For most components, the order is the following: Create the component, set up the component fully, and add the component to a container.

By default, a Scroll pane does not display scrollbars unless they are needed. If the component inside the pane is no larger than the pane itself, the bars won't appear. In the case of components such as text areas, where the component size might increase as the program is used, the bars automatically appear when they're needed and disappear when they're not.

To override this behavior, you can set a policy for a `JScrollBar` component when you create it, using one of several constants in the `ScrollPaneConstants` interface:

- `HORIZONTAL_SCROLLBAR_ALWAYS`
- `HORIZONTAL_SCROLLBAR_AS_NEEDED`
- `HORIZONTAL_SCROLLBAR_NEVER`
- `VERTICAL_SCROLLBAR_ALWAYS`
- `VERTICAL_SCROLLBAR_AS_NEEDED`
- `VERTICAL_SCROLLBAR_NEVER`

These class constants are used with the `JScrollPane(`*`Object`*`, `*`int`*`, `*`int`*`)` constructor, which specifies the component in the pane, the vertical scrollbar policy, and the horizontal scrollbar policy. Here's an example:

Click here to view code image

```
JScrollPane scroller = new JScrollPane(textBox,
    VERTICAL_SCROLLBAR_ALWAYS,
    HORIZONTAL_SCROLLBAR_NEVER);
```

Toolbars

A *toolbar*, created in Swing with the `JToolBar` class, is a container that groups several components into a row or column. These components are most often buttons.

Toolbars are rows or columns of components that group the most commonly used program options. Toolbars often contain buttons and lists and can be

used as an alternative to using pull-down menus or shortcut keys.

Toolbars are horizontal by default, but the orientation can be set explicitly with the `HORIZONTAL` or `VERTICAL` class variables of the `SwingConstants` interface.

Constructor methods include the following:

- `JToolBar()`—Creates a new toolbar
- `JToolBar(int)`—Creates a new toolbar with the specified orientation

After you have created a toolbar, you can add components to it with the toolbar's `add(Object)` method, where *`Object`* represents the component to place on the toolbar.

Many programs that use toolbars enable the user to move the bars. These are called *dockable toolbars* because you can dock them along an edge of the screen, similar to docking a boat. Swing toolbars also can be docked into a new window, separate from the original.

For best results, a dockable `JToolBar` component should be arranged in a container using the `BorderLayout` class, which is a user interface class called a layout manager. A border layout divides a container into five areas: north, south, east, west, and center. Each of the directional components takes up whatever space it needs, and the rest are allocated to the center.

The toolbar should be placed in one of the directional areas of the border layout. The only other area of the layout that can be filled is the center. (Youl learn about layout managers such as border layout during tomorrow's lesson, Day 11, "Arranging Components on a User Interface.")

Figure 10.8 shows a dockable toolbar occupying the south area of a border layout. A text area has been placed in the center.

FIGURE 10.8 A dockable toolbar and a text area.

Listing 10.3 shows the source code used to produce this application.

LISTING 10.3 The Full Text of `FeedBar.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class FeedBar extends JFrame {
 7:
 8:     public FeedBar() {
 9:         super("FeedBar");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         // create icons
13:         ImageIcon loadIcon = new ImageIcon("load.gif");
14:         ImageIcon saveIcon = new ImageIcon("save.gif");
15:         ImageIcon subIcon = new ImageIcon("subscribe.gif");
16:         ImageIcon unsubIcon = new
ImageIcon("unsubscribe.gif");
17:         // create buttons
18:         JButton load = new JButton("Load", loadIcon);
19:         JButton save = new JButton("Save", saveIcon);
20:         JButton sub = new JButton("Subscribe", subIcon);
21:         JButton unsub = new JButton("Unsubscribe",
unsubIcon);
22:         // add buttons to toolbar
23:         JToolBar bar = new JToolBar();
24:         bar.add(load);
25:         bar.add(save);
26:         bar.add(sub);
```

```
27:          bar.add(unsub);
28:          // prepare user interface
29:          JTextArea edit = new JTextArea(8, 40);
30:          JScrollPane scroll = new JScrollPane(edit);
31:          BorderLayout bord = new BorderLayout();
32:          setLayout(bord);
33:          add("North", bar);
34:          add("Center", scroll);
35:          pack();
37:          setVisible(true);
38:      }
39:
40:      private void setLookAndFeel() {
41:          try {
42:              UIManager.setLookAndFeel(
43:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
44:              );
45:              SwingUtilities.updateComponentTreeUI(this);
46:          } catch (Exception e) {
47:              System.err.println("Couldn't use the system "
48:                  + "look and feel: " + e);
49:          }
50:      }
51:
52:      public static void main(String[] arguments) {
53:          FeedBar frame = new FeedBar();
54:      }
55: }
```

This application uses four images to represent the graphics on the buttons—the same graphics used in the IconFrame project yesterday. If you haven't downloaded them yet, they are available on the book's official website at www.java21days.com on the Day 10 page. You also can use graphics from your own computer.

Four `ImageIcon` objects are created from the four graphics in lines 13–16, and then they are used to create buttons in lines 18–21. A `JToolbar` is created in line 23, and the buttons are added to it in lines 24–27.

The toolbar in this application starts at the top edge of the frame, but it can be moved. The component can be grabbed by its handle—the area immediately to the left of the Load button shown in Figure 10.8. If you drag it within the window, you can dock it along different edges of the application window. When you release the toolbar, the application is rearranged using the border

layout manager. You also can drag the toolbar outside the application window.

If the toolbar has been dragged to its own window, when the frame is closed, the toolbar also will close.

Although toolbars are most commonly used with graphical buttons, they can contain textual buttons, combo boxes, and other components.

Progress Bars

Progress bars are components used to show how much time is left before a task is complete.

Progress bars are implemented in Swing through the `JProgressBar` class. Figure 10.9 shows a Java application that uses this component.

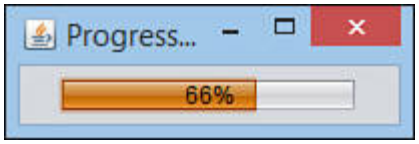


FIGURE 10.9 A progress bar in a frame.

Progress bars are used to track the progress of a task that can be represented numerically. They are created by specifying a minimum and a maximum value that represent the points at which the task is beginning and ending.

Consider a software program that consists of 335 files when it is installed on a computer. This is a good example of a task that can be numerically quantified. The number of files transferred can be used to monitor the progress of the task. The minimum value is 0, and the maximum value is 335.

Constructor methods include the following:

- `JProgressBar()`—Creates a new progress bar
- `JProgressBar(int, int)`—Creates a new progress bar with the specified minimum value and maximum value
- `JProgressBar(int, int, int)`—Creates a new progress bar with the specified orientation, minimum value, and maximum value

The orientation of a progress bar can be established with the `SwingConstants.VERTICAL` and `SwingConstants.HORIZONTAL` class constants. Progress bars are horizontal by default.

You also can set the minimum and maximum values by calling the progress bar's `setMinimum(int)` and `setMaximum(int)` values with the indicated values.

To update a progress bar, you call its `setValue(int)` method with a value indicating how far along the task is at that moment. This value should be somewhere between the minimum and maximum values established for the bar. The following example tells the `install` progress bar in the previous example of a software installation how many files have been uploaded thus far:

Click here to view code image

```
int filesDone = getNumberOfFiles();
install.setValue(filesDone);
```

In this example, the `getNumberOfFiles()` method represents some code that would be used to keep track of how many files have been copied so far during the installation. When this value is passed to the progress bar by the `setValue()` method, the bar is immediately updated to represent the percentage of the task that has been completed.

Progress bars often include a text label in addition to the graphic of an empty box filling up. This label displays the percentage of the task that has been completed. You can set it up for a bar by calling the `setStringPainted(boolean)` method with a value of `true`. A `false` argument turns off this label.

Listing 10.4 contains ProgressMonitor, the application shown at the beginning of this section in Figure 10.9.

LISTING 10.4 The Full Text of `ProgressMonitor.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class ProgressMonitor extends JFrame {
 7:
 8:     JProgressBar current;
 9:     JTextArea out;
10:     JButton find;
```

```
11:     int num = 0;
12:
13:     public ProgressMonitor() {
14:         super("Progress Monitor");
15:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16:         setLookAndFeel();
17:         setSize(205, 68);
18:         setLayout(new FlowLayout());
19:         current = new JProgressBar(0, 2000);
20:         current.setValue(0);
21:         current.setStringPainted(true);
22:         add(current);
23:     }
24:
25:     public void iterate() {
26:         while (num < 2000) {
27:             current.setValue(num);
28:             try {
29:                 Thread.sleep(1000);
30:             } catch (InterruptedException e) { }
31:             num += 95;
32:         }
33:     }
34:
35:     private void setLookAndFeel() {
36:         try {
37:             UIManager.setLookAndFeel(
38:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
39:             );
40:             SwingUtilities.updateComponentTreeUI(this);
41:         } catch (Exception e) {
42:             System.err.println("Couldn't use the system "
43:                 + "look and feel: " + e);
44:         }
45:     }
46:
47:     public static void main(String[] arguments) {
48:         ProgressMonitor frame = new ProgressMonitor();
49:         frame.setVisible(true);
50:         frame.iterate();
51:     }
52: }
```

The ProgressMonitor application uses a progress bar to track the value of the `num` variable. The progress bar is created in line 19 with a minimum value of 0 and a maximum value of 2,000.

The `iterate()` method in lines 25–33 loops while `num` is less than 2,000 and increases `num` by 95 each iteration. The progress bar's `setValue()` method is called in line 27 of the loop with `num` as an argument, causing the bar to use that value when charting progress.

Using a progress bar is a way to make a program more user-friendly when it will be busy for more than a few seconds. Software users like progress bars because they estimate how much more time something will take.

Progress bars also provide another essential piece of information: proof that the program is still running and has not crashed.

Menus

One way you can enhance a frame's usability is to give it a menu bar, a series of pull-down menus used to perform tasks. Menus often duplicate the same tasks you could accomplish by using buttons and other user interface components, giving users two ways to get work done.

Menus in Java are supported by three components that work in conjunction with each other:

- `JMenuItem`—An item on a menu
- `JMenu`—A drop-down menu that contains one or more `JMenuItem` components, other interface components, and separators—lines displayed between items
- `JMenuBar`—A container that holds one or more `JMenu` components and displays their names

A `JMenuItem` component is like a button and can be set up using the same constructor methods as a `JButton` component. Call it with `JMenuItem(`*`String`*`)` for a text item, `JMenuItem(`*`Icon`*`)` for an item that displays a graphics file, or `JMenuItem(`*`String`*`, `*`Icon`*`)` for both.

The following statements create seven menu items:

Click here to view code image

```
JMenuItem j1 = new JMenuItem("Open");
JMenuItem j2 = new JMenuItem("Save");
JMenuItem j3 = new JMenuItem("Save as Template");
JMenuItem j4 = new JMenuItem("Page Setup");
JMenuItem j5 = new JMenuItem("Print");
JMenuItem j6 = new JMenuItem("Use as Default Message Style");
JMenuItem j7 = new JMenuItem("Close");
```

A `JMenu` container holds all the menu items for a drop-down menu. To create it, call the `JMenu(`*`String`*`)` constructor with the name of the menu as an argument. This name appears on the menu bar.

After you have created a `JMenu` container, call its `add(`*`JMenuItem`*`)` to add a menu item to it. New items are placed at the end of the menu.

The item you put on a menu doesn't have to be a menu item. Call the `add(`*`Component`*`)` method with a user interface component as the argument. One that often appears on a menu is a check box (the `JCheckBox` class in Java).

To add a line separator to the end of the menu, call the `addSeparator()` method. Separators often are used to visually group several related items on a menu.

You also can add text to a menu that serves as a label of some kind. Call the `add(`*`String`*`)` method with the text as an argument.

Using the seven menu items from the preceding example, the following statements create a menu and fill it with all those items and three separators:

```
JMenu m1 = new JMenu("File");
m1.add(j1);
m1.add(j2);
m1.add(j3);
m1.addSeparator();
m1.add(j4);
m1.add(j5);
m1.addSeparator();
m1.add(j6);
m1.addSeparator();
m1.add(j7);
```

A `JMenuBar` container holds one or more `JMenu` containers and displays each of their names. The most common place to see a menu bar is directly below an application's title bar.

To create a menu bar, call the `JMenuBar()` constructor method with no arguments. Add menus to the end of a bar by calling its `add(`*`JMenu`*`)` method.

After you have created all your items, added them to menus, and added the menus to a bar, you're ready to add them to a frame. Call the frame's `setJMenuBar(`*`JMenuBar`*`)` method.

The following statement finishes the current example by creating a menu bar, adding a menu to it, and then placing the bar on a frame called `gui`:

```
JMenuBar bar = new JMenuBar();
bar.add(m7);
gui.setJMenuBar(bar);
```

Figure 10.10 shows what this menu looks like on an otherwise empty menu bar.

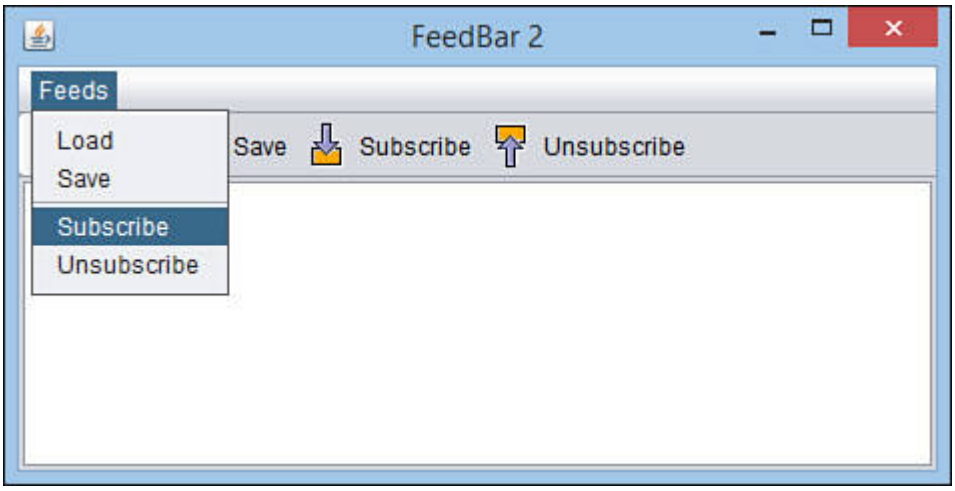


FIGURE 10.10 A frame with a menu bar.

Although you can open and close a menu and select items, nothing happens in response. You'll learn how to receive user input for this component and others during Day 12, "Responding to User Input."

Listing 10.5 is an expanded version of the FeedBar project, adding a menu bar that holds one menu and four individual items.

LISTING 10.5 The Full Text of `FeedBar2.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class FeedBar2 extends JFrame {
 7:
 8:     public FeedBar2() {
 9:         super("FeedBar 2");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
11:          setLookAndFeel();
12:          // create icons
13:          ImageIcon loadIcon = new ImageIcon("load.gif");
14:          ImageIcon saveIcon = new ImageIcon("save.gif");
15:          ImageIcon subIcon = new ImageIcon("subscribe.gif");
16:          ImageIcon unsubIcon = new
ImageIcon("unsubscribe.gif");
17:          // create buttons
18:          JButton load = new JButton("Load", loadIcon);
19:          JButton save = new JButton("Save", saveIcon);
20:          JButton sub = new JButton("Subscribe", subIcon);
21:          JButton unsub = new JButton("Unsubscribe",
unsubIcon);
22:          // add buttons to toolbar
23:          JToolBar bar = new JToolBar();
24:          bar.add(load);
25:          bar.add(save);
26:          bar.add(sub);
27:          bar.add(unsub);
28:          // create menu
29:          JMenuItem j1 = new JMenuItem("Load");
30:          JMenuItem j2 = new JMenuItem("Save");
31:          JMenuItem j3 = new JMenuItem("Subscribe");
32:          JMenuItem j4 = new JMenuItem("Unsubscribe");
33:          JMenuBar menubar = new JMenuBar();
34:          JMenu menu = new JMenu("Feeds");
35:          menu.add(j1);
36:          menu.add(j2);
37:          menu.addSeparator();
38:          menu.add(j3);
39:          menu.add(j4);
40:          menubar.add(menu);
41:          // prepare user interface
42:          JTextArea edit = new JTextArea(8, 40);
43:          JScrollPane scroll = new JScrollPane(edit);
44:          BorderLayout bord = new BorderLayout();
45:          setLayout(bord);
46:          add("North", bar);
47:          add("Center", scroll);
48:          setJMenuBar(menubar);
49:          pack();
50:          setVisible(true);
51:      }
52:
53:      private void setLookAndFeel() {
54:          try {
55:              UIManager.setLookAndFeel(
56:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
57:              );
```

```
58:             SwingUtilities.updateComponentTreeUI(this);
59:         } catch (Exception e) {
60:             System.err.println("Couldn't use the system "
61:                 + "look and feel: " + e);
62:         }
63:     }
64:
65:     public static void main(String[] arguments) {
66:         FeedBar2 frame = new FeedBar2();
67:     }
68: }
```

This application creates the menu bar in lines 29–40 and adds it to the frame in line 48. Figure 10.10 shows the application running.

Tabbed Panes

Tabbed panes, a group of stacked panels in which only one panel can be viewed at a time, are implemented in Swing by the `JTabbedPane` class.

To view a panel, you click the tab that contains its name. Tabs can be arranged horizontally across the top or bottom of the component or vertically along the left or right side.

Tabbed panes are created with the following three constructor methods:

- `JTabbedPane()`—Creates a vertical tabbed pane along the top that does not scroll
- `JTabbedPane(int)`—Creates a tabbed pane that does not scroll and has the specified placement
- `JTabbedPane(int, int)`—Creates a tabbed pane with the specified placement (first argument) and scrolling policy (second argument)

The placement of a tabbed pane is the position where its tabs are displayed in relation to the panels. Use one of four class variables as the argument to the constructor: `JTabbedPane.TOP`, `JTabbedPane.BOTTOM`, `JTabbedPane.LEFT`, or `JTabbedPane.RIGHT`.

The scrolling policy determines how tabs will be displayed when there are more tabs than the interface can hold. A tabbed pane that does not scroll displays extra tabs on their own line, which can be set up using the `JTabbedPane.WRAP_TAB_LAYOUT` class variable. A tabbed pane that

scrolls displays scrolling arrows beside the tabs. This can be set up with `JTabbedPane.SCROLL_TAB_LAYOUT`.

After you create a tabbed pane, you can add components to it by calling the pane's `addTab(String, Component)` method. The `String` argument will be used as the tab's label. The second argument is the component that will make up one of the tabs on the pane. It's common but not required to use a `JPanel` object for this purpose.

The `TabPanels` application in Listing 10.6 displays a pane with five tabs, each holding its own panel.

LISTING 10.6 The Full Text of `TabPanels.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class TabPanels extends JFrame {
 7:
 8:     public TabPanels() {
 9:         super("Tabbed Panes");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         setSize(480, 218);
13:         JPanel mainSettings = new JPanel();
14:         JPanel advancedSettings = new JPanel();
15:         JPanel privacySettings = new JPanel();
16:         JPanel emailSettings = new JPanel();
17:         JPanel securitySettings = new JPanel();
18:         JTabbedPane tabs = new JTabbedPane();
19:         tabs.addTab("Main", mainSettings);
20:         tabs.addTab("Advanced", advancedSettings);
21:         tabs.addTab("Privacy", privacySettings);
22:         tabs.addTab("E-mail", emailSettings);
23:         tabs.addTab("Security", securitySettings);
24:         add(tabs);
25:         setVisible(true);
26:     }
27:
28:     private void setLookAndFeel() {
29:         try {
30:             UIManager.setLookAndFeel(
31:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
```

```
dFeel"
32:                 );
33:                 SwingUtilities.updateComponentTreeUI(this);
34:             } catch (Exception e) {
35:                 System.err.println("Couldn't use the system "
36:                     + "look and feel: " + e);
37:             }
38:         }
39:
40:         public static void main(String[] arguments) {
41:             TabPanels frame = new TabPanels();
42:         }
43: }
```

Five panels are created in lines 13–17. A tabbed pane is created, and then the panels are added to each tab in lines 18–23. Each panel can hold its own user interface components.

Figure 10.11 shows the application running.

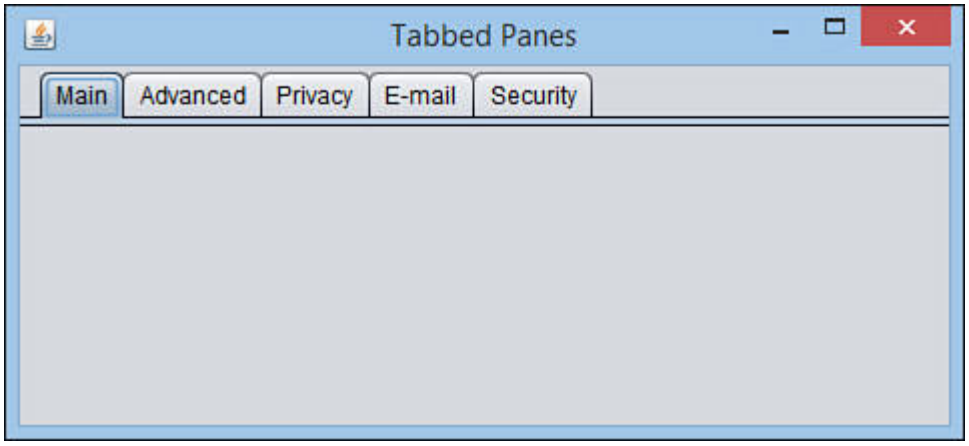


FIGURE 10.11 A tabbed pane with five tabs.

Note

The `TOP`, `BOTTOM`, `LEFT`, and `RIGHT` constants used in `JTabbedPane` constructors are part of the `SwingConstants` interface in the `javax.swing` package. The interface contains a set of integer constants that help position and align components.

Summary

You now know how to paint a user interface onto a Java application window using the components of the Swing package.

Swing includes classes for many of the buttons, bars, lists, and fields you would expect to see on a program. It also includes more advanced components, such as sliders, dialog boxes, progress bars, and menu bars. You implement interface components by creating an instance of their class and adding it to a container such as a frame. You use the container's `add()` method or a similar method specific to the container, such as the tabbed pane's `addTab()` method.

Today, you developed components and added them to an interface. During the next two days, you will learn about two tasks required to make a graphical interface usable. You will see how to arrange components to form a whole interface and how to receive input from a user through these components.

Swing offers a lot more user interface components than the ones covered today and yesterday. Visit Oracle's Javadoc site at http://docs.oracle.com/javase/8/docs/api and click the `javax.swing` package link to explore these classes further.

Q&A

Q Can an application be created without Swing?

A Certainly. Swing is just an expansion on the Abstract Windowing Toolkit, so you could use only AWT classes to design your interface and receive input from a user. But there's no comparison between Swing's capabilities and those offered by the AWT. With Swing, you can use many more components, control them in more sophisticated ways, and count on better performance and more reliability.

Java includes an alternative to Swing called JavaFX that had the original design goal of replacing it in a future release of the language. JavaFX is designed for the creation of desktop applications, mobile applications, and Rich Internet Applications (RIA). JavaFX includes its own user interface components and support for animation, 3D graphics, and HTML 5. The popularity of Android for Java mobile development has undercut some of the rationale for JavaFX, so it appears Swing will continue to be the most widely implemented GUI platform in Java.

Other user interface libraries also extend or compete with Swing. One of the most popular is the Standard Widget Toolkit (SWT), an open source GUI library created by the Eclipse project. The SWT offers

components that appear and behave like the interface components offered by each operating system. For more information, visit www.eclipse.org/swt.

Q In the Slider application, what does the `pack()` statement do?

A Every interface component has a preferred size, although this often is disregarded by the layout manager used to arrange the component within a container. Calling a frame or window's `pack()` method causes it to be resized to fit the preferred size of the components it contains. Because the Slider application does not set a size for the frame, calling `pack()` sets it to an adequate size before the frame is displayed.

Q When I try to create a tabbed pane, all that appears are the tabs—the panels themselves are not visible. What can I do to correct this?

A Tabbed panes won't work correctly until their contents have been fully set up with components inside them. If a tab's panes are empty, nothing appears below or beside the tabs. Make sure that the panels you are putting into the tabs display all their components.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which user interface component is common in software installation programs?

   A. Sliders

   B. Progress bars

   C. Dialog boxes

2. Which Java package includes a class for clickable buttons?

   A. `java.awt` (Abstract Windowing Toolkit)

   B. `javax.swing` (Swing)

   C. Both

3. Which user interface component can be picked up and moved around?

A. `JSlider`

B. `JToolBar`

C. Both

Answers

1. B. Progress bars are useful when used to display the progress of a file-copying or file-extracting activity.
2. C. Swing duplicates all the simple user interface components included in the Abstract Windowing Toolkit.
3. B. The toolbar can be dragged to the top, right, left, or bottom of the interface and also out of the interface.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
import java.awt.*;
import javax.swing.*;

public class AskFrame extends JFrame {
    public AskFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JSlider value = new JSlider(0, 255, 100);
        add(value);
        setSize(450, 150);
        setVisible(true);
        super();
    }

    public static void main(String[] arguments) {
        AskFrame af = new AskFrame();
    }
}
```

What will happen when you attempt to compile and run this source code?

A. It compiles without error and runs correctly.

B. It compiles without error but does not display anything in the frame.

C. It does not compile because of the `super()` statement.

D. It does not compile because of the `add()` statement.

The answer is available on the book's website at www.java21days.com. Visit the Day 10 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create an input dialog box that can be used to set the title of the frame that loaded the dialog box.
2. Create a modified version of the ProgressMonitor application that also displays the value of the `num` variable in a text field.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 11. Arranging Components on a User Interface

If designing a graphical user interface (GUI) were comparable to painting, currently you could produce only one kind of art: abstract expressionism. You can put components on an interface, but you can't control where they go.

To arrange the components of a user interface in Java, you must use a set of classes called layout managers.

Today, you will learn how to use layout managers to arrange components in an interface. You will take advantage of the flexibility of Java's graphical user interface capabilities, which were designed to be presentable on the many platforms that support the language.

You also will learn how to put several layout managers to work on the same interface. This approach is for the many times when one layout manager doesn't suit the exact interface you seek to design.

Basic Interface Layout

As you learned yesterday, a GUI designed with Swing is a fluid thing. Resizing a window can wreak havoc on your interface, because components move to places on a container that you might not have intended.

This fluidity is a necessary part of Java's support for different platforms, where there are subtle differences in how each platform displays things such as buttons, scrollbars, and other parts of a user interface.

With some programming languages, a component's location on a window is precisely defined by its (x,y) coordinate. Some Java development tools allow similar control over an interface through the use of their own windowing classes (and there's a way to do that in Java).

When using Swing, a programmer gains more control over the layout of an interface by using layout managers.

The platform-independent nature of Swing provides flexibility at the cost of slower performance and a user interface look and feel that doesn't closely match the native look and feel of the operating system.

Laying Out an Interface

A layout manager determines how components will be arranged when they are added to a container.

The default layout manager for panels is the `FlowLayout` class. This class lets components flow from left to right in the order in which they are added to a container. When there's no more room, a new row of components begins immediately below the first, and the left-to-right order continues.

Java includes a bunch of general-purpose layout managers: `BorderLayout`, `BoxLayout`, `CardLayout`, `FlowLayout`, and `GridLayout`. To create a layout manager for a container, first call its constructor to create an instance of the class, as in this example:

Click here to view code image

```
FlowLayout flo = new FlowLayout();
```

After you create a layout manager, you designate it as the layout manager for a container by using the container's `setLayout()` method. The layout manager must be set before any components are added to the container. If no layout manager is specified, the container's default layout is used. The default is `FlowLayout` for panels and `BorderLayout` for frames.

The following statements represent the starting point for a frame that uses a layout manager to control the arrangement of all the components that will be added to the frame:

Click here to view code image

```
import java.awt.*;
import javax.swing.*;

public class Starter extends JFrame {

    public Starter() {
        super("Example Frame");
        FlowLayout manager = new FlowLayout();
        setLayout(manager);
        // add components here
    }
}
```

After the layout manager is set, you can start adding components to the container it manages. For some of the layout managers, such as

FlowLayout, the order in which components are added is significant. You'll see this as you work with each of the managers.

Flow Layout

The FlowLayout class in the java.awt package is the simplest layout manager. It lays out components in rows in the same way that words are laid out on a page in English—from left to right until there's no more room at the right edge, and then on to the left edge on the next row.

By default, the components in each row are centered when you use the FlowLayout () constructor with no arguments. If you want the components to be aligned along the left or right edge of the container, you can use the FlowLayout.LEFT or FlowLayout.RIGHT class variable as the constructor's only argument, as in the following statement:

Click here to view code image

```
FlowLayout righty = new FlowLayout(FlowLayout.RIGHT);
```

The FlowLayout.CENTER class variable specifies a centered alignment for components.

Note

If you need to align components for a non-English-speaking audience where left-to-right order does not make sense, you can use the FlowLayout.LEADING and FlowLayout.TRAILING variables. They set justification to the side of either the first component in a row or the last, respectively.

The Alphabet application, shown in Listing 11.1, displays six buttons arranged by the flow layout manager. Because the FlowLayout.LEFT class variable is used in the FlowLayout() constructor, the components are lined up along the left side of the application window. Create this application in NetBeans in the com.java21days package.

LISTING 11.1 The Full Text of Alphabet.java

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class Alphabet extends JFrame {
 8:
 9:     public Alphabet() {
10:         super("Alphabet");
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         setLookAndFeel();
13:         setSize(360, 120);
14:         FlowLayout lm = new FlowLayout(FlowLayout.LEFT);
15:         setLayout(lm);
16:         JButton a = new JButton("Alibi");
17:         JButton b = new JButton("Burglar");
18:         JButton c = new JButton("Corpse");
19:         JButton d = new JButton("Deadbeat");
20:         JButton e = new JButton("Evidence");
21:         JButton f = new JButton("Fugitive");
22:         add(a);
23:         add(b);
24:         add(c);
25:         add(d);
26:         add(e);
27:         add(f);
28:         setVisible(true);
29:     }
30:
31:     private void setLookAndFeel() {
32:         try {
33:             UIManager.setLookAndFeel(
34:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
35:             );
36:             SwingUtilities.updateComponentTreeUI(this);
37:         } catch (Exception exc) {
38:             System.err.println("Couldn't use the system "
39:                 + "look and feel: " + exc);
40:         }
41:     }
42:
43:     public static void main(String[] arguments) {
44:         Alphabet frame = new Alphabet();
45:     }
46: }
```

Figure 11.1 shows the application running.

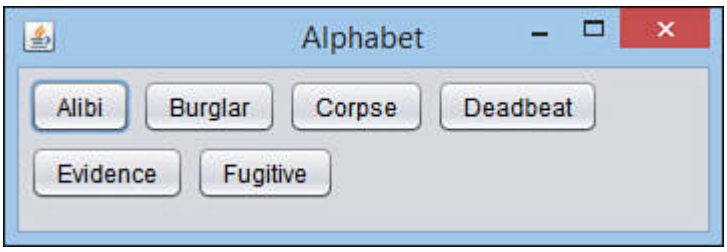


FIGURE 11.1 Six buttons arranged by a flow layout manager.

The Alphabet application creates a flow layout manager in line 14 and sets it to manage the frame in line 15. The buttons added to the frame in lines 22–27 are arranged by this manager.

The manager uses the default gap of 5 pixels between each component on a row and a gap of 5 pixels between each row. You can change the horizontal and vertical gap between components with some extra arguments to the `FlowLayout()` constructor or by calling flow layout's `setVgap(int)` and `setHgap(int)` methods with the desired vertical or horizontal gap.

The `FlowLayout(int, int, int)` constructor takes the following three arguments, in order:

- The alignment, which must be one of five class variables of `FlowLayout`: `CENTER`, `LEFT`, `RIGHT`, `LEADING`, or `TRAILING`
- The horizontal gap between components, in pixels
- The vertical gap, in pixels

The following constructor creates a flow layout manager with centered components, a horizontal gap of 30 pixels, and a vertical gap of 10 pixels:

Click here to view code image

```
FlowLayout flo = new FlowLayout(FlowLayout.CENTER, 30, 10);
```

Box Layout

The next layout manager can be used to stack components from top to bottom or from left to right. Box layout, managed by the `BoxLayout` class in the `javax.swing` package, improves on flow layout by ensuring that components always line up vertically or horizontally, regardless of how their container is resized.

A box layout manager must be created with two arguments to its constructor: the container it will manage and a class variable that sets up vertical or horizontal alignment.

The alignment, specified with class variables of the `BoxLayout` class, can be `X_AXIS` for left-to-right horizontal alignment or `Y_AXIS` for top-to-bottom vertical alignment.

The following code sets up a panel to use vertical box layout:

Click here to view code image

```
JPanel optionPane = new JPanel();
BoxLayout box = new BoxLayout(optionPane, BoxLayout.Y_AXIS);
optionPane.setLayout(box);
```

Components added to the container will line up on the specified axis and will be displayed at their preferred sizes. In horizontal alignment, the box layout manager attempts to give each component the same height. In vertical alignment, the manager attempts to give each one the same width.

The Stacker application, shown in Listing 11.2, contains a panel of buttons arranged with box layout. Create it in NetBeans in the `com.java21days` package.

LISTING 11.2 The Full Text of `Stacker.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class Stacker extends JFrame {
 7:     public Stacker() {
 8:         super("Stacker");
 9:         setSize(430, 150);
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         // create top panel
13:         JPanel commandPane = new JPanel();
14:         BoxLayout horizontal = new BoxLayout(commandPane,
15:             BoxLayout.X_AXIS);
16:         commandPane.setLayout(horizontal);
17:         JButton subscribe = new JButton("Subscribe");
18:         JButton unsubscribe = new JButton("Unsubscribe");
19:         JButton refresh = new JButton("Refresh");
20:         JButton save = new JButton("Save");
21:         commandPane.add(subscribe);
22:         commandPane.add(unsubscribe);
```

```
23:          commandPane.add(refresh);
24:          commandPane.add(save);
25:          // create bottom panel
26:          JPanel textPane = new JPanel();
27:          JTextArea text = new JTextArea(4, 70);
28:          JScrollPane scrollPane = new JScrollPane(text);
29:          // put them together
30:          FlowLayout flow = new FlowLayout();
31:          setLayout(flow);
32:          add(commandPane);
33:          add(scrollPane);
34:          setVisible(true);
35:      }
36:
37:      private void setLookAndFeel() {
38:          try {
39:              UIManager.setLookAndFeel(
40:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
41:              );
42:              SwingUtilities.updateComponentTreeUI(this);
43:          } catch (Exception exc) {
44:              System.err.println("Couldn't use the system "
45:                  + "look and feel: " + exc);
46:          }
47:      }
48:
49:      public static void main(String[] arguments) {
50:          Stacker st = new Stacker();
51:      }
52: }
```

When the class is compiled and run, the output should resemble Figure 11.2.

FIGURE 11.2 A user interface with buttons arranged with the box layout manager.

This application creates a `JPanel` container named `commandPane` in line 13, creates a box layout manager associated with that pane in lines 14–15, and sets that manager for the panel in line 16.

The panel of buttons along the top edge of the interface is stacked horizontally. If the second argument to the box layout constructor was `BoxLayout.Y_AXIS`, the buttons would be arranged vertically instead.

Grid Layout

The grid layout manager arranges components into a grid of vertical columns and horizontal rows like the days on a 12-month calendar. Components are added first to the top row of the grid, beginning with the leftmost grid cell and continuing to the right. When all the cells in the top row are full, the next component is added to the leftmost cell in the second row of the grid—if there *is* a second row—and so on.

Grid layout managers are created with the `GridLayout` class, which belongs to the `java.awt` package. Two arguments are sent to the `GridLayout` constructor: the number of rows and the number of columns in the grid.

The following statement creates a grid layout manager with 10 rows and 3 columns:

Click here to view code image

```
GridLayout gr = new GridLayout(10, 3);
```

As with flow layout, you can specify a vertical and horizontal gap between components with two extra arguments (or by calling the `setHgap()` or `setVgap()` methods). The following statement creates a grid layout with 10 rows and 3 columns, a horizontal gap of 5 pixels, and a vertical gap of 8 pixels:

Click here to view code image

```
GridLayout gr2 = new GridLayout(10, 3, 5, 8);
```

The default gap between components arranged in grid layout is 0 pixels in both vertical and horizontal directions.

For the next project, create the Bunch application in the `com.java21days` package, which is shown in Listing 11.3. The program creates a grid with 3 rows, 3 columns, and a 10-pixel gap between components in both the vertical and horizontal directions.

LISTING 11.3 The Full Text of `Bunch.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class Bunch extends JFrame {
 8:
 9:     public Bunch() {
10:         super("Bunch");
11:         setSize(260, 260);
12:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13:         setLookAndFeel();
14:         JPanel pane = new JPanel();
15:         GridLayout family = new GridLayout(3, 3, 10, 10);
16:         pane.setLayout(family);
17:         JButton marcia = new JButton("Marcia");
18:         JButton carol = new JButton("Carol");
19:         JButton greg = new JButton("Greg");
20:         JButton jan = new JButton("Jan");
21:         JButton alice = new JButton("Alice");
22:         JButton peter = new JButton("Peter");
23:         JButton cindy = new JButton("Cindy");
24:         JButton mike = new JButton("Mike");
25:         JButton bobby = new JButton("Bobby");
26:         pane.add(marcia);
27:         pane.add(carol);
28:         pane.add(greg);
29:         pane.add(jan);
30:         pane.add(alice);
31:         pane.add(peter);
32:         pane.add(cindy);
33:         pane.add(mike);
34:         pane.add(bobby);
35:         add(pane);
36:         setVisible(true);
37:     }
38:
39:     private void setLookAndFeel() {
40:         try {
41:             UIManager.setLookAndFeel(
42:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
43:             );
44:             SwingUtilities.updateComponentTreeUI(this);
45:         } catch (Exception exc) {
46:             System.err.println("Couldn't use the system "
47:                 + "look and feel: " + exc);
```

```
48:             }
49:         }
50:
51:         public static void main(String[] arguments) {
52:             Bunch frame = new Bunch();
53:         }
54: }
```

Figure 11.3 shows this application.

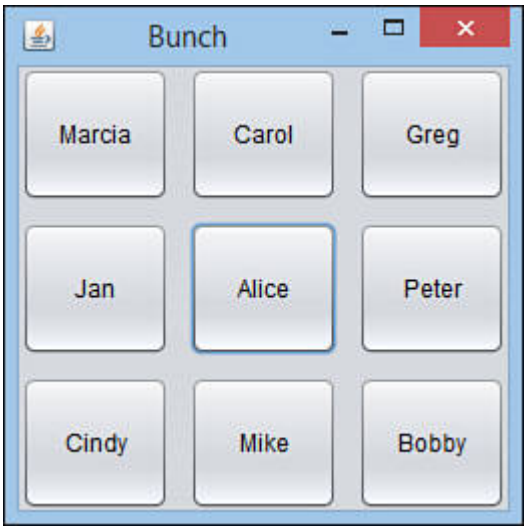


FIGURE 11.3 Nine buttons arranged in a 3×3 grid layout.

The Bunch application displays nine buttons in a grid. The buttons are added to a pane in lines 26–34, and the pane is added to the frame in line 35.

One thing to note about the buttons in Figure 11.3 is that they expanded to fill the space available to them in each cell. This is an important difference between grid layout and some of the other layout managers, which display components at a much smaller size by using the preferred size of those components.

Border Layout

The layout managers introduced so far have been fairly simple. The next one employs a more complex arrangement called border layout.

This layout is created by using the `BorderLayout` class in the `java.awt` package, which divides a container into five sections: north, south, east, west, and center. The five areas in Figure 11.4 show how these sections are arranged.

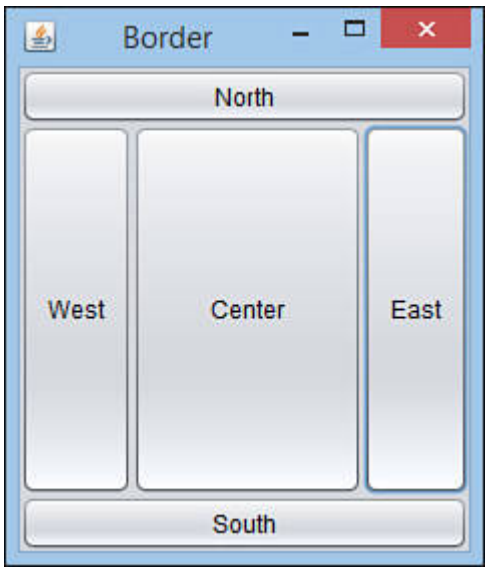


FIGURE 11.4 Components arranged by a border layout manager.

In border layout, the components represented by the four compass points fill their sections, and the center component gets all the space that's left over. Ordinarily, this results in an arrangement with a large central component and four smaller components around it. The preferred sizes of the components are not followed by the layout manager.

A border layout is created with either the `BorderLayout()` or `BorderLayout(int, int)` constructors. The first constructor creates a border layout with no gap between any of the components. The second constructor uses arguments to specify the horizontal gap and vertical gap, in that order, and `setVgap()` and `setHgap()` also are available.

After you create a border layout and set it up as a container's layout manager, components are added using a call to the `add()` method that's different from the ones seen previously:

```
add(Component, String)
```

The first argument is the component that should be added to the container.

The second argument is a `BorderLayout` class variable that indicates the region of the border layout to which the component should be assigned. The class variables `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER` can be used for this argument.

The following statement adds a button called `quitButton` to the north portion of a border layout:

Click here to view code image

```
JButton quitButton = new JButton("quit");
add(quitButton, BorderLayout.NORTH);
```

The Border application, shown in Listing 11.4, creates the GUI shown earlier in Figure 11.4. Create the `Border` class in the `com.java21days` package.

LISTING 11.4 The Full Text of `Border.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class Border extends JFrame {
 7:
 8:     public Border() {
 9:         super("Border");
10:         setSize(240, 280);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         setLookAndFeel();
13:         setLayout(new BorderLayout());
14:         JButton nButton = new JButton("North");
15:         JButton sButton = new JButton("South");
16:         JButton eButton = new JButton("East");
17:         JButton wButton = new JButton("West");
18:         JButton cButton = new JButton("Center");
19:         add(nButton, BorderLayout.NORTH);
20:         add(sButton, BorderLayout.SOUTH);
21:         add(eButton, BorderLayout.EAST);
22:         add(wButton, BorderLayout.WEST);
23:         add(cButton, BorderLayout.CENTER);
24:         setVisible(true);
25:     }
26:
27:     private void setLookAndFeel() {
28:         try {
29:             UIManager.setLookAndFeel(
30:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
31:             );
32:             SwingUtilities.updateComponentTreeUI(this);
33:         } catch (Exception exc) {
34:             System.err.println("Couldn't use the system "
```

```
35:                     + "look and feel: " + exc);
36:             }
37:         }
38:
39:         public static void main(String[] arguments) {
40:             Border frame = new Border();
41:         }
42: }
```

The Border application is a frame that sets its layout manager in a new way in line 13. The call to the `new BorderLayout()` constructor returns a `BorderLayout` object, which then becomes the argument to the `setLayout()` method.

Line 13 is equivalent to the following two statements:

Click here to view code image

```
BorderLayout bl = new BorderLayout();
setLayout(bl);
```

The advantage of the technique employed in line 13 is that there's no need to create a variable and assign the `BorderLayout` object to it. That object's never needed after the layout manager is designated for the frame.

The application creates the five buttons in lines 14–18 and assigns them to positions in the border layout in lines 19–23.

Tip

When you run the application, increase the window size several times to see how the components respond. As the window becomes larger, the center component grows accordingly. The other components stay the same. This is an advantage of the grid and border layout managers.

Mixing Layout Managers

At this point, you might be wondering how Java's layout managers can be used on the GUIs you want to design for your own programs. Choosing a layout manager is an experience akin to Goldilocks checking out the home of the three bears: This one is too square! This one is too disorganized! This one is too strange!

To find the layout that is just right, you often have to combine more than one manager within the same interface.

You can do so by putting several containers inside a larger container and giving each of the smaller containers its own layout manager.

The container to use for these smaller containers is the panel, which is created from the `JPanel` class in the `javax.swing` package. Panels are simple containers used to group components. Keep in mind two things when working with panels:

- The panel is filled with components before it is put into a larger container.
- The panel has its own layout manager.

Panels are created with a simple call to the constructor of the `JPanel` class, as shown in the following example:

```
JPanel pane = new JPanel();
```

You set the layout method for a panel by calling the `setLayout()` method on that panel. Here's how to create a layout manager and apply it to a `JPanel` object called `pane`:

Click here to view code image

```
FlowLayout flo = new FlowLayout();
pane.setLayout(flo);
```

You add components to a panel by calling the panel's `add()` method, which works the same for panels as it does for other containers.

The following statements create a text field and add it to a `JPanel` object called `pane`:

Click here to view code image

```
JTextField nameField = new JTextField(80);
pane.add(nameField);
```

You'll see several examples of panel use in the rest of today's applications.

As you gain experience with layout managers, you get a feel for which ones to use in specific situations. For instance, border layout is good for putting a status line at the bottom and a toolbar at the top, and grid layout is effective for rows and columns of text fields and labels that take the same size.

Card Layout

A card layout manager differs from the other layout managers because it hides some components from view. A card layout is a group of containers or components displayed one at a time, in the same way that a blackjack dealer reveals one card at a time from a deck. Each container in the group is called a *card*.

If you have used a wizard in an installation program, you have seen card layout. Each step in the installation process has its own card. Often, a Next button advances from one card to the next.

The most common way to use a card layout is to use a panel container for each card. Components are added to the panels first, and then the panels are added to the container that employs card layout.

A card layout is created from the `CardLayout` class in the `java.awt` package with a simple constructor:

Click here to view code image

```
CardLayout cc = new CardLayout();
```

The `setLayout()` method makes this the layout manager for the container, as in the following statement:

```
setLayout(cc);
```

After you set a container to use the card layout manager, you must use the `add(`*`Component`*`, `*`String`*`)` method to add components.

The first argument to the `add()` method specifies the container or component that serves as a card. If it is a container, all components must have been added to it before the card is added.

The second argument is a string that names the card. This can be anything you want to call the card, such as `"Card 1"`, `"Card 2"`, `"Card 3"`, or some other naming scheme.

The following statement adds a panel object named `options` to a container and names this card `"Options Card"`:

```
add(options, "Options Card");
```

When a container using card layout is displayed for the first time, the visible card is the first card added to the container.

You can display subsequent cards by calling the `show()` method of the layout manager, which takes two arguments:

- The container holding all the cards
- The name of the card

The following statement calls the `show ()` method of a card layout manager called `cc`:

```
cc.show(this, "Fact Card");
```

The `this` keyword would be used in a frame governed by card layout. It refers to the object inside which the `cc.show()` statement appears. In this example, "Fact Card" is the name of the card to reveal. A card is added to the container that has been given this name.

When a card is shown, the previously displayed card is hidden automatically. Only one card in a card layout can be shown at a time.

In a program that uses the card layout manager, a card change generally is triggered by a user's action. For example, in an installation program, a user could choose a folder where the program should be saved and click the Next button to see the next card.

Using Card Layout in an Application

The next project demonstrates both card layout and the use of different layout managers within the same GUI.

The `SurveyWizard` class is a panel that implements a wizard interface: a series of simple questions accompanied by a Next button that is used to see the subsequent question. The last question has a Finish button instead and is shown in Figure 11.5.

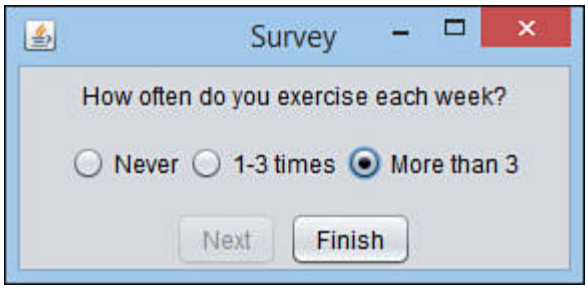


FIGURE 11.5 Using a card layout for a wizard-style interface.

The easiest way to implement a card-based layout is to use panels. This project uses several panels:

- The `SurveyWizard` class is a panel that holds all the cards.
- The `SurveyPanel` helper class is a panel that holds one card.
- Each `SurveyPanel` object contains three panels stacked on top of each other.

The `SurveyWizard` and `SurveyPanel` classes are both panels, the easiest component to use when working with card layout. Each card is created as a panel and is added to a containing panel that will be used to show them in sequence.

This takes place in the `SurveyWizard()` constructor, using two instance variables, a card layout manager, and an array of three `SurveyPanel` objects:

Click here to view code image

```
SurveyPanel[] ask = new SurveyPanel[3];
CardLayout cards = new CardLayout();
```

The constructor sets the class to use the layout manager, creates each `SurveyPanel` object, and then adds it to the class:

Click here to view code image

```
setLayout(cards);
String question1 = "What is your gender?";
String[] resp1 = { "female", "male", "not telling" };
ask[0] = new SurveyPanel(question1, resp1, 2);
add(ask[0], "Card 0");
```

Each `SurveyPanel` object is created with three arguments to the constructor: the text of the question, an array of possible responses, and the element number of the default answer.

In the preceding code, the question “What is your gender?” has the responses “female”, “male”, and “not telling”. The response at position 2, “not telling”, is set as the default.

The `SurveyPanel` constructor uses a label component to hold the question and an array of radio buttons to hold the responses:

Click here to view code image

```
SurveyPanel(String ques, String[] resp, int def) {
    question = new JLabel(ques);
    response = new JRadioButton[resp.length];
    // more to come
}
```

The class uses grid layout to arrange its components into a grid with three vertical columns and one horizontal row. Each component placed in the grid is a panel.

First, a panel is created to hold the question label:

Click here to view code image

```
JPanel sub1 = new JPanel();
JLabel quesLabel = new JLabel(ques);
sub1.add(quesLabel);
```

The default layout for panels, flow layout with centered alignment, determines the placement of the label on the panel.

Next, a panel is created to hold the possible responses. A `for` loop iterates through the string array that holds the text of each response. This text is used to create a radio button. The second argument of the `JRadioButton()` constructor determines whether it is selected. This is implemented with the following code:

Click here to view code image

```
JPanel sub2 = new JPanel();
for (int i = 0; i < resp.length; i++) {
    if (def == i) {
        response[i] = new JRadioButton(resp[i], true);
    } else {
        response[i] = new JRadioButton(resp[i], false);
    }
    group.add(response[i]);
    sub2.add(response[i]);
}
```

The last panel holds the Next and Finish buttons:

```
JPanel sub3 = new JPanel();
nextButton.setEnabled(true);
sub3.add(nextButton);
finalButton.setEnabled(false);
sub3.add(finalButton);
```

Now that the three panels have been fully set up, they are added to the `SurveyPanel` interface, which completes the work of the constructor method:

Click here to view code image

```
GridLayout grid = new GridLayout(3, 1);
setLayout(grid);
add(sub1);
add(sub2);
add(sub3);
```

There's one extra wrinkle in the `SurveyPanel` class—a method that enables the Finish button and disables the Next button when the last question has been reached:

Click here to view code image

```
void setFinalQuestion(boolean finalQuestion) {
    if (finalQuestion) {
        nextButton.setEnabled(false);
        finalButton.setEnabled(true);
    }
}
```

In a user interface that uses card layout, the display of each card usually takes place in response to an action by the user.

These actions are called events, which are covered on Day 12, "Responding to User Input."

A brief preview demonstrates how the `SurveyPanel` class is equipped to handle button clicks.

The class implements `ActionListener`, an interface in the `java.awt.event` package:

Click here to view code image

```
public class SurveyWizard extends JPanel implements
ActionListener {
    // more to come
}
```

This interface indicates that the class can respond to action events, which represent button clicks, menu choices, and similar user input.

Next, each button's `addActionListener(`*`Object`*`)` method is called:

Click here to view code image

```
ask[0].nextButton.addActionListener(this);
ask[0].finalButton.addActionListener(this);
```

Listeners are classes that monitor specific kinds of user input. The argument to `addActionListener()` is the class that's looking for action events.

Using `this` as the argument indicates that the `SurveyPanel` class handles this job.

The `ActionListener` interface includes only one method:

Click here to view code image

```
public void actionPerformed(Action evt) {
    // more to come
}
```

This method is called when a component being listened to generates an action event. In the `SurveyPanel` class, this happens whenever a button is clicked.

In `SurveyPanel`, this method uses an instance variable that keeps track of which card to display:

```
int currentCard = 0;
```

Every time a button is clicked and the `actionPerformed()` method is called, this variable is incremented, and the card layout manager's `show(`*`Container`*`, `*`String`*`)` method is called to display a new card. If the last card has been displayed, the Finish button is disabled.

Listing 11.5 shows the full `SurveyWizard` class with the complete `actionPerformed()` method. Create a new empty Java file in NetBeans called `SurveyWizard`, assigning it to the `com.java21days` package.

LISTING 11.5 The Full Text of `SurveyWizard.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class SurveyWizard extends JPanel implements
ActionListener {
 8:     int currentCard = 0;
 9:     CardLayout cards = new CardLayout();
10:     SurveyPanel[] ask = new SurveyPanel[3];
11:
12:     public SurveyWizard() {
13:         super();
```

```
14:         setSize(240, 140);
15:         setLayout(cards);
16:         // set up survey
17:         String question1 = "What is your gender?";
18:         String[] resp1 = { "female", "male", "not telling"
};
19:         ask[0] = new SurveyPanel(question1, resp1, 2);
20:         String question2 = "What is your age?";
21:         String[] resp2 = { "Under 25", "25-34", "35-54",
22:             "Over 54" };
23:         ask[1] = new SurveyPanel(question2, resp2, 1);
24:         String question3 = "How often do you exercise each
week?";
25:         String[] resp3 = { "Never", "1-3 times", "More than
3" };
26:         ask[2] = new SurveyPanel(question3, resp3, 1);
27:         ask[2].setFinalQuestion(true);
28:         addListeners();
29:     }
30:
31:     private void addListeners() {
32:         for (int i = 0; i < ask.length; i++) {
33:             ask[i].nextButton.addActionListener(this);
34:             ask[i].finalButton.addActionListener(this);
35:             add(ask[i], "Card " + i);
36:         }
37:     }
38:
39:     public void actionPerformed(ActionEvent evt) {
40:         currentCard++;
41:         if (currentCard >= ask.length) {
42:             System.exit(0);
43:         }
44:         cards.show(this, "Card " + currentCard);
45:     }
46: }
47:
48: class SurveyPanel extends JPanel {
49:     JLabel question;
50:     JRadioButton[] response;
51:     JButton nextButton = new JButton("Next");
52:     JButton finalButton = new JButton("Finish");
53:
54:     SurveyPanel(String ques, String[] resp, int def) {
55:         super();
56:         setSize(160, 110);
57:         question = new JLabel(ques);
58:         response = new JRadioButton[resp.length];
59:         JPanel sub1 = new JPanel();
60:         ButtonGroup group = new ButtonGroup();
```

```
61:          JLabel quesLabel = new JLabel(ques);
62:          sub1.add(quesLabel);
63:          JPanel sub2 = new JPanel();
64:          for (int i = 0; i < resp.length; i++) {
65:              if (def == i) {
66:                  response[i] = new JRadioButton(resp[i],
true);
67:              } else {
68:                  response[i] = new JRadioButton(resp[i],
false);
69:              }
70:              group.add(response[i]);
71:              sub2.add(response[i]);
72:          }
73:          JPanel sub3 = new JPanel();
74:          nextButton.setEnabled(true);
75:          sub3.add(nextButton);
76:          finalButton.setEnabled(false);
77:          sub3.add(finalButton);
78:          GridLayout grid = new GridLayout(3, 1);
79:          setLayout(grid);
80:          add(sub1);
81:          add(sub2);
82:          add(sub3);
83:      }
84:
85:      void setFinalQuestion(boolean finalQuestion) {
86:          if (finalQuestion) {
87:              nextButton.setEnabled(false);
88:              finalButton.setEnabled(true);
89:          }
90:      }
91: }
```

The `SurveyWizard` class is a `JPanel` component that creates a card layout manager as an instance variable in line 9 and assigns it to the panel in line 15. This class lacks a `main()` method, so it must be added to another program's user interface to be tested.

The SurveyFrame application, shown in Listing 11.6, contains a frame that displays a survey panel. Create it in NetBeans (package `com.java21days`).

LISTING 11.6 The Full Text of `SurveyFrame.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class SurveyFrame extends JFrame {
 7:     public SurveyFrame() {
 8:         super("Survey");
 9:         setSize(290, 140);
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         SurveyWizard wiz = new SurveyWizard();
13:         add(wiz);
14:         setVisible(true);
15:     }
16:
17:     private void setLookAndFeel() {
18:         try {
19:             UIManager.setLookAndFeel(
20:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
21:             );
22:             SwingUtilities.updateComponentTreeUI(this);
23:         } catch (Exception exc) {
24:             System.err.println("Couldn't use the system "
25:                 + "look and feel: " + exc);
26:         }
27:     }
28:
29:     public static void main(String[] arguments) {
30:         SurveyFrame surv = new SurveyFrame();
31:     }
32: }
```

A `SurveyWizard` object is created in line 12 and is added to the frame in line 13. The running application was shown earlier in Figure 11.5.

Cell Padding and Insets

By default, no components have extra space around them (which is easiest to see in components that fill their cells).

The horizontal and vertical gaps that appear when you create a new layout manager are used to determine the amount of space between components in a panel. *Insets*, however, are used to determine the amount of space around the panel itself. The `Insets` class includes values for the top, bottom, left, and right insets, which then are used when the panel is drawn.

Insets determine the amount of space between the edges of a panel and that panel's components.

The following statement creates an `Insets` object that specifies 20 pixels of insets above and below and 13 pixels to the left and right:

Click here to view code image

```
Insets whitespace = new Insets(20, 13, 20, 13);
```

You can establish insets in any container by overriding its `getInsets()` method and returning an `Insets` object, as in this example:

Click here to view code image

```
public Insets getInsets() {
    return new Insets(10, 30, 10, 30);
}
```

Summary

When it comes to designing a user interface in Java, you've seen today that abstract expressionism goes only so far. Getting the desired user interface layout in a Swing application requires the use of layout managers.

These managers require some adjustment for people who are used to more precise control over where components appear on an interface.

You now know how to use the five layout managers and panels. As you work with Swing, you'll find that it can approximate any kind of interface through the use of nested containers and different layout managers.

After you master the development of a user interface in Java, your programs can offer an interface that works on multiple platforms without modification.

Q&A

Q I really dislike working with layout managers; they're either too simplistic or too complicated. Even with a lot of tinkering, I can never get my user interface to look like I want it to. All I want to do is define the sizes of my components and put them at an (x,y) position on the screen. Can I do this?

A It's possible, but problematic. Java was designed in such a way that a program's GUI could run equally well on different platforms and with different screen resolutions, fonts, screen sizes, and the like. Relying on

pixel coordinates can cause a program that looks good on one platform to be unusable on others. Layout disasters such as components overlapping each other or getting cut off by the edge of a container may result. Layout managers, by dynamically placing elements on the screen, get around these problems. Although there might be some differences in the end results on different platforms, they are less likely to be catastrophic.

If none of that is persuasive, here's how to ignore my advice: Set the content pane's layout manager with `null` as the argument. Create a `Rectangle` object (from the `java.awt` package) with the (x,y) position, width, and height of the component as arguments. Finally, call the component's `setBounds(Rectangle)` method with that rectangle as the argument.

The following application displays a 300×300-pixel frame with a Click Me button at the (x,y) position 10, 10 that is 120 pixels wide by 30 pixels tall:

Click here to view code image

```
import java.awt.*;
import javax.swing.*;

public class Absolute extends JFrame {
    public Absolute() {
        super("Example");
        setSize(300, 300);
        setLayout(null);
        JButton myButton = new JButton("Click Me");
        myButton.setBounds(new Rectangle(10, 10, 120, 30));
        add(myButton);
        setVisible(true);
    }

    public static void main(String[] arguments) {
        Absolute ex = new Absolute();
    }
}
```

You can find out more about `setBounds()` in the `Component` class. You can find the documentation for the Java class library at http://docs.oracle.com/javase/8/docs/api.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What is the default layout manager for a panel in Java?

   A. None

   B. `BorderLayout`

   C. `FlowLayout`

2. Which layout manager uses a compass direction or a reference to the center when adding a component to a container?

   A. `BorderLayout`

   B. `MapLayout`

   C. `FlowLayout`

3. If you want to create an installation wizard that has multiple steps, what layout manager should you use?

   A. `GridLayout`

   B. `CardLayout`

   C. `BorderLayout`

Answers

1. C. To keep a panel from using flow layout, you can set its layout manager to `null`.
2. A. Border layout has class variables `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`.
3. B. Card layout enables components to be stacked like cards and displayed one at a time, making it well-suited to implement a wizard.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
import java.awt.*;
import javax.swing.*;

public class ThreeButtons extends JFrame {
    public ThreeButtons() {
        super("Program");
        setSize(350, 225);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JButton alpha = new JButton("Alpha");
        JButton beta = new JButton("Beta");
        JButton gamma = new JButton("Gamma");
        // answer goes here
        add(alpha);
        add(beta);
        add(gamma);
        pack();
        setVisible(true);
    }

    public static void main(String[] arguments) {
        ThreeButtons b3 = new ThreeButtons();
    }
}
```

Which statement should replace `// answer goes here` to make the frame display all three buttons side by side?

A. `content.setLayout(null);`

B. `content.setLayout(new FlowLayout());`

C. `content.setLayout(new GridLayout(3,1));`

D. `content.setLayout(new BorderLayout());`

The answer is available on the book's website at www.java21days.com. Visit the Day 11 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a user interface that displays a calendar for a single month, including headings for the seven days of the week and a title for the month across the top.
2. Create an interface that incorporates more than one layout manager.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 12. Responding to User Input

Designing a Java program with a graphical user interface (GUI) isn't very useful if the user can't do anything to it. To make the program completely functional, you must make the interface receptive to user events.

Swing handles events with a set of interfaces called event listeners. You create a listener object and associate it with the user interface component being monitored.

Today, you will learn how to add listeners of all kinds to your Swing programs, including those that handle action events, mouse events, and other interaction.

When you're finished, you will have created a full Java application using the Swing set of classes.

Event Listeners

If a class wants to respond to a user event in Java, it must implement the interface that deals with the events. This interface is not the same thing as a GUI. The interface is an abstract type that defines methods a class must implement.

Interfaces that handle user events are called event listeners.

Each listener handles a specific kind of event.

The `java.awt.event` package contains all the basic event listeners, as well as the objects that represent specific events. These listener interfaces are some of the most useful:

- `ActionListener`—*Action events*, which are generated when a user performs an action on a component, such as clicking a button
- `AdjustmentListener`—*Adjustment events*, which are generated when a component is adjusted, such as when a scrollbar is moved
- `FocusListener`—*Keyboard focus events*, which are generated when a component such as a text field gains or loses the focus
- `ItemListener`—*Item events*, which are generated when an item such as a check box is changed

- `KeyListener`—*Keyboard events*, which occur when a user enters text using the keyboard
- `MouseListener`—*Mouse events*, which are generated by mouse clicks, a mouse entering a component's area, and a mouse leaving a component's area
- `MouseMotionListener`—*Mouse movement events*, which track all movement by a mouse over a component
- `WindowListener`—*Window events*, which are generated when a window is maximized, minimized, moved, or closed

Just as a Java class can implement multiple interfaces, a class that takes user input can implement as many listeners as needed. The `implements` keyword in the class declaration is followed by the name of the interface. If more than one interface has been implemented, their names are separated by commas.

The following class is declared to handle both action and text events:

Click here to view code image

```
public class Suspense extends JFrame implements ActionListener,
     TextListener {
    // body of class
}
```

To refer to these event listener interfaces in your programs, you can import them individually or use an `import` statement with a wildcard to make the entire package available:

```
import java.awt.event.*;
```

Setting Up Components

When you make a class an event listener, you have set up a specific type of event to be heard by that class. However, the event won't be heard unless you follow up with a second step: You must add a matching listener to the GUI component. That listener generates the events when the component is used.

After a component is created, you can call one (or more) of the following methods on the component to associate a listener with it:

- `addActionListener()`—`JButton`, `JCheckBox`, `JComboBox`, `JTextField`, `JRadioButton`, and `JMenuItem`

components

- `addFocusListener()`—All Swing components
- `addItemListener()`—`JButton`, `JCheckBox`, `JComboBox`, and `JRadioButton` components
- `addKeyListener()`—All Swing components
- `addMouseListener()`—All Swing components
- `addMouseMotionListener()`—All Swing components
- `addTextListener()`—`JTextField` and `JTextArea` components
- `addWindowListener()`—`JWindow` and `JFrame` components

Caution

Modifying a component after adding it to a container is an easy mistake to make in a Java program. You must add listeners to a component and handle any other configuration before the component is added to any containers; otherwise, these settings are disregarded when the program is run.

The following example creates a `JButton` object and associates an action event listener with it:

Click here to view code image

```
JButton zap = new JButton("Zap");
zap.addActionListener(this);
```

All the listener adding methods take one argument: the object that is listening for events of that kind. Using `this` indicates that the current class is the event listener. You could specify a different object, as long as its class implements the right listener interface.

Event-Handling Methods

When you associate an interface with a class, the class must contain methods that implement every method in the interface.

In the case of event listeners, the windowing system calls each method automatically when the corresponding user event takes place.

The `ActionListener` interface has only one method: `actionPerformed()`. All classes that implement `ActionListener` must have a method with the following structure:

Click here to view code image

```
public void actionPerformed(ActionEvent event) {
    // handle event here
}
```

If only one component in your program's GUI has a listener for action events, you will know that this `actionPerformed()` method is called only in response to an event generated by that component.

This makes it simpler to write the `actionPerformed()` method. All the method's code responds to that component's user event.

But when more than one component has an action event listener, you must use the method's `ActionEvent` argument to figure out which component was used and act accordingly in your program. You can use this object to discover details about the component that generated the event.

`ActionEvent` and all other event objects are part of the `java.awt.event` package.

Every event-handling method is sent an event object of some kind. You can use the object's `getSource()` method to determine which component sent the event, as in the following example:

Click here to view code image

```
public void actionPerformed(ActionEvent event) {
    Object source = evt.getSource();
}
```

The object returned by the `getSource()` method can be compared to components by using the == operator. The following statements extend the preceding example to handle user clicks on buttons named `quitButton` and `sortRecords`:

```
if (source == quitButton) {
    quit();
}
if (source == sortRecords) {
    sort();
}
```

The `quit()` method is called if the `quitButton` object generated the event, and the `sort()` method is called if the `sortRecords` button generated the event.

Many event-handling methods call a different method for each kind of event or component. This makes the event-handling method easier to read. In addition, if a class has more than one event-handling method, each one can call the same methods to get work done.

Java's `instanceof` operator can be used in an event-handling method to determine the class of component that generated the event. The following example can be used in a program with one button and one text field, each of which generates an action event:

Click here to view code image

```
void actionPerformed(ActionEvent event) {
    Object source = event.getSource();
    if (source instanceof JTextField) {
        calculateScore();
    } else if (source instanceof JButton) {
        quit();
    }
}
```

If the event-generating component belongs to the `JTextField` class, the `calculateScore()` method is called. If the component belongs to `JButton`, the `quit()` method is called instead.

The TitleBar application, shown in Listing 12.1, displays a frame with two `JButton` components, which are used to change the text on the frame's title bar. Create a new empty Java file called `TitleBar`, assign it the package `com.java21days`, and enter the class's source code.

LISTING 12.1 The Full Text of `TitleBar.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.event.*;
 4: import javax.swing.*;
 5: import java.awt.*;
 6:
 7: public class TitleBar extends JFrame implements
ActionListener {
```

```
 8:     JButton b1;
 9:     JButton b2;
10:
11:     public TitleBar() {
12:         super("Title Bar");
13:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:         setLookAndFeel();
15:         b1 = new JButton("Rosencrantz");
16:         b2 = new JButton("Guildenstern");
17:         b1.addActionListener(this);
18:         b2.addActionListener(this);
19:         FlowLayout flow = new FlowLayout();
20:         setLayout(flow);
21:         add(b1);
22:         add(b2);
23:         pack();
24:         setVisible(true);
25:     }
26:
27:     public void actionPerformed(ActionEvent evt) {
28:         Object source = evt.getSource();
29:         if (source == b1) {
30:             setTitle("Rosencrantz");
31:         } else if (source == b2) {
32:             setTitle("Guildenstern");
33:         }
34:         repaint();
35:     }
36:
37:     private void setLookAndFeel() {
38:         try {
39:             UIManager.setLookAndFeel(
40:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
41:             );
42:             SwingUtilities.updateComponentTreeUI(this);
43:         } catch (Exception exc) {
44:             System.err.println("Couldn't use the system "
45:                 + "look and feel: " + exc);
46:         }
47:     }
48:
49:     public static void main(String[] arguments) {
50:         TitleBar frame = new TitleBar();
51:     }
52: }
```

After you run this application with the Java Virtual Machine (JVM), the program's interface should resemble Figure 12.1.

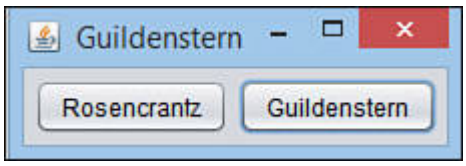


FIGURE 12.1 The TitleBar application.

Only 13 lines are needed to respond to action events in this application:

- Line 3 imports the `java.awt.event` package.
- Line 7 indicates the class implements the `ActionListener` interface.
- Lines 17–18 add action listeners to both `JButton` objects.
- Lines 27–35 respond to action events that occur from the two `JButton` objects. The `evt` object's `getSource()` method determines the event's source. If it is equal to the `b1` button, the frame's title is set to `Rosencrantz`; if it is equal to `b2`, the title is set to `Guildenstern`. A call to `repaint()` is needed so that the frame is redrawn after any title change that might have occurred in the method.

Working with Methods

The following sections detail the structure of each event-handling method and the methods that can be used within them.

In addition to the methods described, the `getSource()` method can be used on any event object to determine which object generated the event.

Action Events

Action events occur when a user completes an action using components such as buttons, check boxes, menu items, text fields, and radio buttons.

A class must implement the `ActionListener` interface to handle these events. In addition, the `addActionListener()` method must be called on each component that should generate an action event—unless you want to ignore that component's action events.

The `actionPerformed (`*`ActionEvent`*`)` method is the only method of the `ActionListener` interface. It takes the following form:

Click here to view code image

```
public void actionPerformed(ActionEvent event) {
   // ...
}
```

In addition to the `getSource()` method, you can use the `getActionCommand()` method on the `ActionEvent` object to discover more information about the event's source.

By default, the action command is the text associated with the component, such as the label on a button. You also can set a different action command for a component by calling its `setActionCommand(`*`String`*`)` method. The string argument should be the action command's desired text.

The following statements create a button and menu item and give both of them the action command "Sort Files":

Click here to view code image

```
JButton sort = new JButton("Sort");
JMenuItem menuSort = new JMenuItem("Sort");
sort.setActionCommand("Sort Files");
menuSort.setActionCommand("Sort Files");
```

Tip

Action commands are useful in a program in which more than one component should cause the same thing to happen. By giving both components the same action command, you can handle them with the same code in an event-handling method.

Focus Events

Focus events occur when any component gains or loses input focus on a GUI. *Focus* describes the component that is active for keyboard input. If one of the fields has the focus (in a user interface with several editable text fields), the cursor blinks in the field. Any text entered goes into this component.

Focus applies to all components that can receive input. You can give a component the focus by calling its `requestFocus()` method with no arguments, as in this example:

Click here to view code image

```
JButton ok = new JButton("OK");
ok.requestFocus();
```

To handle a focus event, a class must implement the `FocusListener` interface, which has two methods: `focusGained(`*`FocusEvent`*`)` and `focusLost(`*`FocusEvent`*`)`. They take the following forms:

Click here to view code image

```
public void focusGained(FocusEvent event) {
   // ...
}

public void focusLost(FocusEvent event) {
   // ...
}
```

To determine which object gained or lost the focus, the `getSource()` method can be called on the `FocusEvent` object sent as an argument to the two methods.

Listing 12.2 contains Calculator, a Java application that displays the sum of two numbers. Focus events are used to determine when the sum needs to be recalculated. In NetBeans create a new Java file with the name `Calculator` and package name `com.java21days` with the source code of this listing.

LISTING 12.2 The Full Text of `Calculator.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.event.*;
 4: import javax.swing.*;
 5: import java.awt.*;
 6:
 7: public class Calculator extends JFrame implements
FocusListener {
 8:     JTextField value1, value2, sum;
 9:     JLabel plus, equals;
10:
11:     public Calculator() {
12:         super("Add Two Numbers");
13:         setSize(350, 90);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         setLookAndFeel();
16:         FlowLayout flow = new FlowLayout(FlowLayout.CENTER);
17:         setLayout(flow);
18:         // create components
```

```
19:         value1 = new JTextField("0", 5);
20:         plus = new JLabel("+");
21:         value2 = new JTextField("0", 5);
22:         equals = new JLabel("=");
23:         sum = new JTextField("0", 5);
24:         // add listeners
25:         value1.addFocusListener(this);
26:         value2.addFocusListener(this);
27:         // set up sum field
28:         sum.setEditable(false);
29:         // add components
30:         add(value1);
31:         add(plus);
32:         add(value2);
33:         add(equals);
34:         add(sum);
35:         setVisible(true);
36:     }
37:
38:     public void focusGained(FocusEvent event) {
39:         try {
40:             float total = Float.parseFloat(value1.getText())
+
41:                 Float.parseFloat(value2.getText());
42:             sum.setText("" + total);
43:         } catch (NumberFormatException nfe) {
44:             value1.setText("0");
45:             value2.setText("0");
46:             sum.setText("0");
47:         }
48:     }
49:
50:     public void focusLost(FocusEvent event) {
51:         focusGained(event);
52:     }
53:
54:     private void setLookAndFeel() {
55:         try {
56:             UIManager.setLookAndFeel(
57:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
58:             );
59:             SwingUtilities.updateComponentTreeUI(this);
60:         } catch (Exception exc) {
61:             System.err.println("Couldn't use the system "
62:                 + "look and feel: " + exc);
63:         }
64:     }
65:
66:     public static void main(String[] arguments) {
```

```
67:          Calculator frame = new Calculator();
68:      }
69: }
```

Figure 12.2 shows the application.

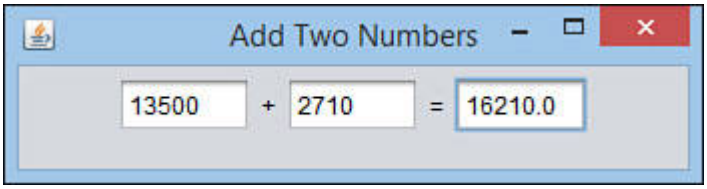


FIGURE 12.2 The Calculator application.

In this application, focus listeners are added to the first two text fields, `value1` and `value2`, and the class implements the `FocusListener` interface.

The `focusGained()` method is called whenever either of these fields gains the input focus (lines 38–48). In this method, the sum is calculated by adding the values in the other two fields. If either field contains an invalid value, such as a string, a `NumberFormatException` is thrown, and all three fields are reset to "0".

The `focusLost()` method accomplishes the same behavior by calling `focusGained()` with the focus event as an argument.

One thing to note about this application is that event-handling behavior is not required to collect numeric input in a text field. This is taken care of automatically by any component in which text input is received.

Item Events

Item events occur when an item is selected or deselected on components such as buttons, check boxes, or radio buttons. A class must implement the `ItemListener` interface to handle these events.

The `itemStateChanged(`*`ItemEvent`*`)` method is the only method in the `ItemListener` interface. It takes the following form:

Click here to view code image

```
void itemStateChanged(ItemEvent event) {
   // ...
}
```

To determine in which item the event occurred, the `getItem()` method can be called on the `ItemEvent` object.

You also can see whether the item was selected or deselected by using the `getStateChange ()` method. This method returns an integer that equals either the class variable `ItemEvent.DESELECTED` or `ItemEvent.SELECTED`.

The FormatChooser application, shown in in Listing 12.3, illustrates the use of item events, displaying information about a selected combo box item in a label. Create it with NetBeans as an empty Java file with the class name `FormatChooser` and package name `com.java21days`.

LISTING 12.3 The Full Text of `FormatChooser.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class FormatChooser extends JFrame implements
ItemListener {
 8:     String[] formats = { "(choose format)", "Atom", "RSS
0.92",
 9:         "RSS 1.0", "RSS 2.0" };
10:     String[] descriptions = {
11:         "Atom weblog and syndication format",
12:         "RSS syndication format 0.92 (Netscape)",
13:         "RSS/RDF syndication format 1.0 (RSS/RDF)",
14:         "RSS syndication format 2.0 (UserLand)"
15:     };
16:     JComboBox formatBox = new JComboBox();
17:     JLabel descriptionLabel = new JLabel("");
18:
19:     public FormatChooser() {
20:         super("Syndication Format");
21:         setSize(420, 150);
22:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23:         setLayout(new BorderLayout());
24:         for (int i = 0; i < formats.length; i++) {
25:             formatBox.addItem(formats[i]);
26:         }
27:         formatBox.addItemListener(this);
28:         add(BorderLayout.NORTH, formatBox);
```

```
29:          add(BorderLayout.CENTER, descriptionLabel);
30:          setVisible(true);
31:      }
32:
33:      public void itemStateChanged(ItemEvent event) {
34:          int choice = formatBox.getSelectedIndex();
35:          if (choice > 0) {
36:               descriptionLabel.setText(descriptions[choice-
1]);
37:          }
38:      }
39:
40:      public Insets getInsets() {
41:          return new Insets(50, 10, 10, 10);
42:      }
43:
44:      private static void setLookAndFeel() {
45:          try {
46:              UIManager.setLookAndFeel(
47:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
48:              );
49:          } catch (Exception exc) {
50:              System.err.println("Couldn't use the system "
51:                  + "look and feel: " + exc);
52:          }
53:      }
54:
55:      public static void main(String[] arguments) {
56:          FormatChooser.setLookAndFeel();
57:          FormatChooser fc = new FormatChooser();
58:      }
59: }
```

This application extends the combo box example from Day 9, "Working with Swing." Figure 12.3 shows how it looks after a choice has been made.

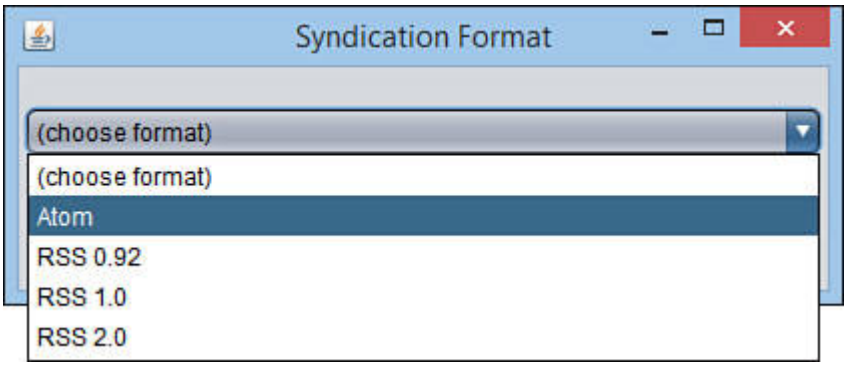


FIGURE 12.3 The output of the FormatChooser application.

The application creates a combo box from an array of strings and adds an item listener to the component (lines 24–27). Item events are received by the `itemStateChanged(ItemEvent)` method (lines 33–38), which changes a label's text based on the index number of the selected item. Index 1 corresponds with `"Atom"`, 2 with `"RSS 0.92"`, 3 with `"RSS 1.0"`, and 4 with `"RSS 2.0"`.

Key Events

Key events occur when a key is pressed on the keyboard. Any component can generate these events, and a class must implement the `KeyListener` interface to support them.

The `KeyListener` interface has three methods: `keyPressed(KeyEvent)`, `keyReleased(KeyEvent)`, and `keyTyped(KeyEvent)`. They take the following forms:

Click here to view code image

```
public void keyPressed(KeyEvent event) {
    // ...
}

public void keyReleased(KeyEvent event) {
    // ...
}

public void keyTyped(KeyEvent event) {
    // ...
}
```

`KeyEvent`'s `getKeyChar()` method returns the character of the key associated with the event. If no Unicode character can be represented by the key, `getKeyChar()` returns a character value equal to the class variable `KeyEvent.CHAR_UNDEFINED`.

For a component to generate key events, it must be able to receive the input focus. Text fields, text areas, and other components that accept keyboard input support this capability automatically. For other components, such as labels and panels, the `setFocusable(boolean)` method should be called with an argument of `true`, as in the following code:

```
JPanel pane = new JPanel();
pane.setFocusable(true);
```

Mouse Events

Mouse events are generated by a mouse click, a mouse entering a component's area, or a mouse leaving the area. Any component can generate these events, which are implemented by a class through the `MouseListener` interface, which has five methods:

- `mouseClicked(`*`MouseEvent`*`)`
- `mouseEntered(`*`MouseEvent`*`)`
- `mouseExited(`*`MouseEvent`*`)`
- `mousePressed(`*`MouseEvent`*`)`
- `mouseReleased(`*`MouseEvent`*`)`

Each method takes the same basic form as `mouseReleased(`*`MouseEvent`*`)`:

Click here to view code image

```
public void mouseReleased(MouseEvent event) {
    // ...
}
```

The following methods can be used on `MouseEvent` objects:

- `getClickCount()`—Returns as an integer the number of times the mouse was clicked
- `getPoint()`—Returns as a `Point` object the (x,y) coordinate within the component where the mouse was clicked
- `getX()`—Returns the x position
- `getY()`—Returns the y position

Mouse Motion Events

Mouse motion events occur when the mouse is moved over a component. As with other mouse events, any component can generate mouse motion events. A class must implement the `MouseMotionListener` interface to support them.

The `MouseMotionListener` interface has two methods: `mouseDragged(`*`MouseEvent`*`)` and `mouseMoved(`*`MouseEvent`*`)`. They take the following forms:

Click here to view code image

```
public void mouseDragged(MouseEvent event) {
    // ...
}

public void mouseMoved(MouseEvent event) {
    // ...
}
```

Unlike the other event-listener interfaces you have dealt with up to this point, `MouseMotionListener` does not have its own event type. Instead, `MouseEvent` objects are used.

Because of this, you can call the same methods you would for mouse events: `getClick()`, `getPoint()`, `getX()`, and `getY()`.

The next project you will undertake demonstrates how to detect and respond to mouse events. The MousePrank application, shown in Listing 12.4, consists of two classes, `MousePrank` and `PrankPanel`, that implement a user interface button that tries to avoid being clicked.

Create a new empty Java file in NetBeans with the class name MousePrank and package name `com.java21days`; then enter the code shown in Listing 12.4. The techniques demonstrated in this class will be described after you create the application and see how it runs.

LISTING 12.4 The Full Text of `MousePrank.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class MousePrank extends JFrame implements
ActionListener {
 8:     public MousePrank() {
 9:         super("Message");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setSize(420, 220);
12:         BorderLayout border = new BorderLayout();
13:         setLayout(border);
14:         JLabel msg = new JLabel("Click OK to close
program.");
15:         add(BorderLayout.NORTH, msg);
16:         PrankPanel prank = new PrankPanel();
```

```
 17:         prank.ok.addActionListener(this);
 18:         add(BorderLayout.CENTER, prank);
 19:         setVisible(true);
 20:     }
 21:
 22:     public void actionPerformed(ActionEvent event) {
 23:         System.exit(0);
 24:     }
 25:
 26:     public Insets getInsets() {
 27:         return new Insets(40, 10, 10, 10);
 28:     }
 29:
 30:     private static void setLookAndFeel() {
 31:         try {
 32:             UIManager.setLookAndFeel(
 33:                 "com.sun.java.swing.plaf.nimbus.NimbusLookA
ndFeel"
 34:             );
 35:         } catch (Exception exc) {
 36:             System.err.println("Couldn't use the system "
 37:                 + "look and feel: " + exc);
 38:         }
 39:     }
 40:
 41:     public static void main(String[] arguments) {
 42:         MousePrank.setLookAndFeel();
 43:         new MousePrank();
 44:     }
 45: }
 46:
 47: class PrankPanel extends JPanel implements
MouseMotionListener {
 48:     JButton ok = new JButton("OK");
 49:     int buttonX, buttonY, mouseX, mouseY;
 50:     int width, height;
 51:
 52:     PrankPanel() {
 53:         super();
 54:         setLayout(null);
 55:         addMouseMotionListener(this);
 56:         buttonX = 110;
 57:         buttonY = 110;
 58:         ok.setBounds(new Rectangle(buttonX, buttonY,
 59:             70, 20));
 60:         add(ok);
 61:     }
 62:
 63:     public void mouseMoved(MouseEvent event) {
 64:         mouseX = event.getX();
```

```
 65:          mouseY = event.getY();
 66:          width = (int) getSize().getWidth();
 67:          height = (int) getSize().getHeight();
 68:          if (Math.abs((mouseX + 35) - buttonX) < 50) {
 69:              buttonX = moveButton(mouseX, buttonX, width);
 70:              repaint();
 71:          }
 72:          if (Math.abs((mouseY + 10) - buttonY) < 50) {
 73:              buttonY = moveButton(mouseY, buttonY, height);
 74:              repaint();
 75:          }
 76:      }
 77:
 78:      public void mouseDragged(MouseEvent event) {
 79:          // ignore this event
 80:      }
 81:
 82:      private int moveButton(int mouseAt, int buttonAt, int
bord) {
 83:          if (buttonAt < mouseAt) {
 84:              buttonAt--;
 85:          } else {
 86:              buttonAt++;
 87:          }
 88:          if (buttonAt > (bord - 20)) {
 89:              buttonAt = 10;
 90:          }
 91:          if (buttonAt < 0) {
 92:              buttonAt = bord - 80;
 93:          }
 94:          return buttonAt;
 95:      }
 96:
 97:      public void paintComponent(Graphics comp) {
 98:          super.paintComponent(comp);
 99:          ok.setBounds(buttonX, buttonY, 70, 20);
100:      }
101: }
```

The `MousePrank` class is a frame that holds two components arranged with a border layout—the label "Click OK to close this program." and a panel with an OK button on it. Figure 12.4 shows the user interface for this application.

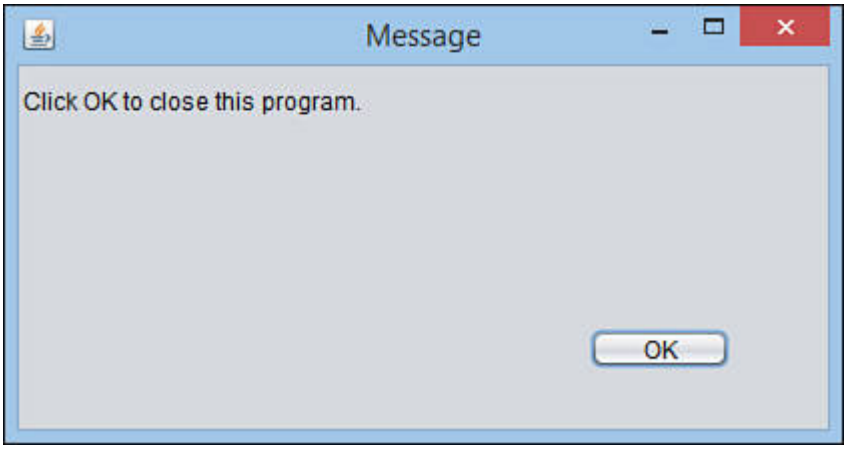


FIGURE 12.4 The running MousePrank application.

Because the button does not behave normally, it is implemented with the `PrankPanel` class, a subclass of `JPanel`. This panel includes a button that is drawn at a specific position on the panel instead of being placed by a layout manager. This technique was described at the end of Day 11, "Arranging Components on a User Interface."

First, the panel's layout manager is set to `null`, which causes it to stop using flow layout as its default manager:

```
setLayout(null);
```

Next, the button is placed on the panel using `setBounds(`*`Rectangle`*`)`, the same method that determines where a frame or window will appear on a desktop.

A `Rectangle` object is created with four arguments: its x position, y position, width, and height. Here's how `PrankPanel` draws the button:

Click here to view code image

```
JButton ok = new JButton("OK");
int buttonX = 110;
int buttonY = 110;
ok.setBounds(new Rectangle(buttonX, buttonY, 70, 20));
```

Creating the `Rectangle` object as the argument to `setBounds()` is more efficient than creating an object with a name and using that object as the argument. You don't need to use the object anywhere else in the class, so it doesn't need a name. The following statements accomplish the same thing in two steps:

Click here to view code image

```
Rectangle box = new Rectangle(buttonX, buttonY, 70, 20);
ok.setBounds(box);
```

The class has instance variables that hold the button's (x,y) position, `buttonX` and `buttonY`. They start out at (110,110) and change whenever the mouse comes within 50 pixels of the center of the button.

You track mouse movements by implementing the `MouseListener` interface and its two methods, `mouseMoved(`*`MouseEvent`*`)` and `mouseDragged(`*`MouseEvent`*`)`.

The panel uses `mouseMoved()` and ignores `mouseDragged()`.

When the mouse moves, a mouse event object's `getX()` and `getY()` methods return its current (x,y) position, which is stored in the instance variables `mouseX` and `mouseY`.

The `moveButton(`*`int`*`, `*`int`*`, `*`int`*`)` method takes three arguments:

- The button's x or y position
- The mouse's x or y position
- The panel's width or height

This method moves the button away from the mouse in either a vertical or horizontal direction, depending on whether it is called with x-coordinates and the panel height or y-coordinates and the width.

After the button's position has moved, the `repaint()` method is called, which causes the panel's `paintComponent(`*`Graphics`*`)` method to be called in lines 97–100.

Every component has a `paintComponent()` method that can be overridden to draw the component. The button's `setBounds()` method displays it at the current (x,y) position in line 99.

Window Events

Window events occur when a user opens or closes a window object, such as a `JFrame` or `JWindow`. Any component can generate these events, and a class must implement the `WindowListener` interface to support them.

The `WindowListener` interface has seven methods:

- `windowActivated(`*`WindowEvent`*`)`
- `windowClosed(`*`WindowEvent`*`)`

- `windowClosing(`*`WindowEvent`*`)`
- `windowDeactivated(`*`WindowEvent`*`)`
- `windowDeiconified(`*`WindowEvent`*`)`
- `windowIconified(`*`WindowEvent`*`)`
- `windowOpened(`*`WindowEvent`*`)`

They all take the same form as the `windowOpened ()` method:

Click here to view code image

```
public void windowOpened(WindowEvent event) {
    // body of method
}
```

The `windowClosing ()` and `windowClosed ()` methods are similar, but one is called as the window is closing and the other is called after it is closed. In fact, you can take action in a `windowClosing()` method to stop the window from being closed.

Using Adapter Classes

A Java class that implements an interface must include all its methods, even if it doesn't plan to do anything in response to some of them.

This requirement can make it necessary to add a lot of empty methods when you're working with an event-handling interface such as `WindowListener`, which has seven methods.

As a convenience, Java offers *adapters*, Java classes that contain empty do-nothing implementations of specific interfaces. By subclassing an adapter class, you can implement only the event-handling methods you need by overriding those methods. The rest inherit those do-nothing methods.

The `java.awt.event` package includes `FocusAdapter`, `KeyAdapter`, `MouseAdapter`, `MouseMotionAdapter`, and `WindowAdapter`. They correspond to the expected listeners for focus, keyboard, mouse, mouse motion, and window events.

Listing 12.5 is a Java application that displays the most recently pressed key, monitoring keyboard events through a subclass of `KeyAdapter`. Enter this source code in a new empty Java class file named `KeyChecker` in NetBeans in the package `com.java21days`.

LISTING 12.5 The Full Text of `KeyChecker.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class KeyChecker extends JFrame {
 8:     JLabel keyLabel = new JLabel("Hit any key");
 9:
10:     public KeyChecker() {
11:         super("Hit a Key");
12:         setSize(300, 200);
13:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:         setLayout(new FlowLayout(FlowLayout.CENTER));
15:         KeyMonitor monitor = new KeyMonitor(this);
16:         setFocusable(true);
17:         addKeyListener(monitor);
18:         add(keyLabel);
19:         setVisible(true);
20:     }
21:
22:     private static void setLookAndFeel() {
23:         try {
24:             UIManager.setLookAndFeel(
25:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
26:             );
27:         } catch (Exception exc) {
28:             System.err.println("Couldn't use the system "
29:                 + "look and feel: " + exc);
30:         }
31:     }
32:
33:     public static void main(String[] arguments) {
34:         KeyChecker.setLookAndFeel();
35:         new KeyChecker();
36:     }
37: }
38:
39: class KeyMonitor extends KeyAdapter {
40:     KeyChecker display;
41:
42:     KeyMonitor(KeyChecker display) {
43:         this.display = display;
44:     }
```

```
45:
46:     public void keyTyped(KeyEvent event) {
47:         display.keyLabel.setText("" + event.getKeyChar());
48:         display.repaint();
49:     }
50: }
```

The KeyChecker application is implemented as a main class of that name and a `KeyMonitor` helper class.

`KeyMonitor` is a subclass of `KeyAdapter`, an adapter class for keyboard events that implements the `KeyListener` interface. In lines 46–49, the `keyTyped` method overrides the same method in `KeyAdapter`, which does nothing.

When a key is pressed, the key is discovered by calling `getKeyChar()` of the user event object. This key becomes the text of the `keyLabel` label in the `KeyChecker` class. This application is shown in Figure 12.5.

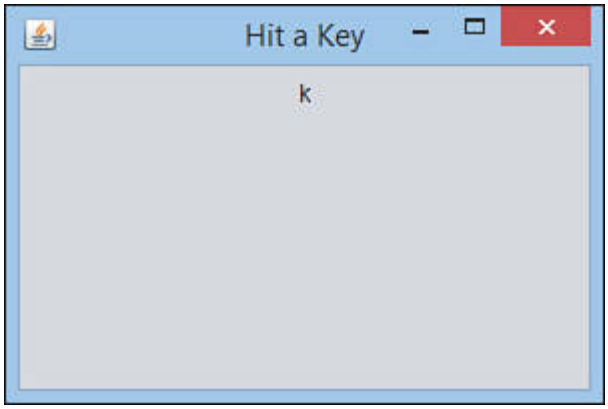


FIGURE 12.5 The running KeyChecker application.

Using Inner Classes

One of the challenges of taking user input in Java is to keep the code as short and simple as possible. The need to implement event listeners and all their methods, even for undesired input, requires a lot of coding.

In the `KeyChecker` application, an adapter class was used to shorten the amount of programming required to handle key events.

A technique to shorten it further would be to use inner classes, which are defined within a class, as if they were a method or variable. An adapter class is created as an inner class in this statement:

Click here to view code image

```
KeyAdapter monitor = new KeyAdapter() {
    public void keyTyped(KeyEvent event) {
        keyLabel.setText("" + event.getKeyChar());
        repaint();
    }
};
```

The `KeyAdapter` object overrides one method, `keyTyped(`*`KeyEvent`*`)`, to receive keyboard input. The `KeyChecker2` class shown in Listing 12.6 has two advantages over its predecessor. As you create it in NetBeans, see if you can figure out what they are.

LISTING 12.6 The Full Text of `KeyChecker2.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class KeyChecker2 extends JFrame {
 8:     JLabel keyLabel = new JLabel("Hit any key");
 9:
10:     public KeyChecker2() {
11:          super("Hit a Key");
12:          setSize(300, 200);
13:          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:          setLayout(new FlowLayout(FlowLayout.CENTER));
15:          KeyAdapter monitor = new KeyAdapter() {
16:              public void keyTyped(KeyEvent event) {
17:                  keyLabel.setText("" + event.getKeyChar());
18:                  repaint();
19:              }
20:          };
21:          setFocusable(true);
22:          addKeyListener(monitor);
23:          add(keyLabel);
24:          setVisible(true);
25:      }
26:
27:      private static void setLookAndFeel() {
28:          try {
29:              UIManager.setLookAndFeel(
30:                  "com.sun.java.swing.plaf.nimbus.NimbusLookA
ndFeel"
31:              );
```

```
32:             } catch (Exception exc) {
33:                 System.err.println("Couldn't use the system "
34:                     + "look and feel: " + exc);
35:             }
36:         }
37:
38:         public static void main(String[] arguments) {
39:             KeyChecker2.setLookAndFeel();
40:             new KeyChecker2();
41:         }
42:     }
```

The application functions identically to the `KeyChecker` version.

The advantages of this version are that it is shorter, not requiring the creation of a separate class, and it does not need to make use of the `this` variable in the inner class to be able to change the label in line 17. The inner class can access the variables and methods of its own class.

Inner classes also can be anonymous, which are objects of the class not assigned to a variable.

The `TitleBar` application developed today, which used action events to change a frame's title in response to button clicks, could be simplified by using anonymous inner classes. An anonymous inner class becomes the argument to the button's `addActionListener()` method, as you can see:

Click here to view code image

```
JButton b1;
b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        setTitle("Rosencrantz");
    }
});
JButton b2;
b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        setTitle("Guildenstern");
    }
});
```

The anonymous inner class is an object that implements the `ActionListener` interface. The object's `actionPerformed()` method is overridden to set the frame's title when the corresponding button is clicked. Because each button has its own listener, it's simpler than using one listener for multiple interface components.

Inner classes look more complicated than separate classes, but they can simplify and shorten your Java code. You will look much further into inner classes during Day 16, "Using Inner Classes and Closures."

Summary

Event handling is added to a GUI in Swing through these fundamental steps:

- A listener interface is added to the class that will contain the event-handling methods.
- A listener is added to each component that will generate the events to handle.
- The methods are added, each with an `EventObject` class as the only argument to the method.
- Methods of that `EventObject` class, such as `getSource()`, are used to learn which component generated the event and what kind of event it was.

When you know these steps, you can work with each of the listener interfaces and event classes. You also can learn about new listeners as they are added to Swing with new components.

Q&A

Q Can a program's event-handling behavior be put into its own class instead of being included with the code that creates the interface?

A It can, and many programmers will tell you that this is a good way to design your programs. Separating interface design from your event-handling code allows you to develop the two separately. This makes it easier to maintain the project; related behavior is grouped and isolated from unrelated behavior.

Q Is there a way to differentiate between the buttons on a `mouseClicked()` event?

A Yes. This feature of mouse events wasn't covered today because right and middle mouse buttons are platform-specific features that are unavailable on some systems where Java programs run.

All mouse events send a `MouseEvent` object to their event-handling methods. Call the object's `getModifiers()` method to receive an

integer value that indicates which mouse button generated the event.

Check the value against three class variables. It equals `MouseEvent.BUTTON1_MASK` if the left button was clicked, `MouseEvent.BUTTON2_MASK` if the middle button was clicked, and `MouseEvent.BUTTON3_MASK` if the right button was clicked. See `MouseTest.java` on the Day 12 page of the book's website at www.java21days.com for an example that implements this technique.

For more information, see the Java class library documentation for the `MouseEvent` class. Visit the web page http://docs.oracle.com/javase/8/docs/api and click the `java.awt.event` hyperlink to view the classes in that package.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. If you use `this` in a method call such as `addActionListener(this)`, what object is being registered as a listener?

   A. An adapter class

   B. The current class

   C. No class

2. What is the benefit of subclassing an adapter class such as `WindowAdapter` (which implements the `WindowListener` interface)?

   A. You inherit all the behavior of that class.

   B. The subclass automatically becomes a listener.

   C. You don't need to implement any `WindowListener` methods you won't be using.

3. What kind of event is generated when you press Tab to leave a text field?

   A. `FocusEvent`

   B. `WindowEvent`

C. `ActionEvent`

Answers

1. B. The current class must implement the correct listener interface and the required methods.
2. C. Because most listener interfaces contain more methods than you will need, using an adapter class as a superclass saves the hassle of implementing empty methods just to implement the interface.
3. A. A user interface component loses focus when the user stops editing that component and moves to a different part of the interface.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

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

public class Expunger extends JFrame implements ActionListener {
    public boolean deleteFile;

    public Expunger() {
        super("Expunger");
        JLabel commandLabel = new JLabel("Do you want to delete
the file?");
        JButton yes = new JButton("Yes");
        JButton no = new JButton("No");
        yes.addActionListener(this);
        no.addActionListener(this);
        setLayout( new BorderLayout() );
        JPanel bottom = new JPanel();
        bottom.add(yes);
        bottom.add(no);
        add("North", commandLabel);
        add("South", bottom);
        pack();
        setVisible(true);
    }
```

```
    public void actionPerformed(ActionEvent evt) {
        JButton source = (JButton) evt.getSource();
        // answer goes here
            deleteFile = true;
        else
            deleteFile = false;
    }

    public static void main(String[] arguments) {
        new Expunger();
    }
}
```

Which of the following statements should replace `// answer goes here` to make the application function correctly?

A. `if (source instanceof JButton)`

B. `if (source.getActionCommand().equals("yes"))`

C. `if (source.getActionCommand().equals("Yes"))`

D. `if source.getActionCommand() == "Yes"`

The answer is available on the book's website at www.java21days.com. Visit the Day 12 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create an application that uses `FocusListener` to ensure that a text field's value is multiplied by –1 and is redisplayed whenever a user changes it to a negative value.
2. Create a calculator that adds or subtracts the contents of two text fields whenever the appropriate button is clicked, displaying the result as a label.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 13. Creating Java2D Graphics

Today, you'll work with Java classes that put the graphics in graphical user interfaces. Java2D is a set of classes that support high-quality, scalable, two-dimensional images, color, and text.

Java2D, which includes classes in the `java.awt` and `javax.swing` packages, can be used for each of these visually appealing tasks:

- Drawing text
- Drawing shapes such as circles and polygons
- Using different fonts, colors, and line widths
- Filling shapes with colors and patterns

The Graphics2D Class

Everything in Java2D begins with the `Graphics2D` class in the `java.awt` package, which represents a graphics context, an environment in which something can be drawn. A `Graphics2D` object can represent a component on a graphical user interface, printer, or another display device.

`Graphics2D` is a subclass of the `Graphics` class that extends and improves its visual capabilities.

Before you can start using the `Graphics2D` class, you need a surface on which to draw.

Several user interface components can act as a canvas for graphical operations, including panels and windows.

As soon as you have a component to use as a canvas, you can draw text, lines, ovals, circles, arcs, rectangles, and other polygons on that object.

One component that's suitable for this purpose is `JPanel` in the `javax.swing` package. This class represents panels in a graphical user interface that can be empty or contain other components.

The following code creates a frame and a panel and then adds the panel to the frame:

Click here to view code image

```
JFrame main = new JFrame("Welcome Screen");
JPanel pane = new JPanel();
```

```
main.add(pane);
```

Like many user interface components in Java, panels have a `paintComponent(Graphics)` method that is called automatically when the component needs to be redisplayed.

Several things could cause `paintComponent()` to be called:

- The graphical user interface containing the component is displayed for the first time.
- A window that was displayed on top of the component is closed.
- The graphical user interface containing the component is resized.

By creating a subclass of `JPanel`, you can override the panel's `paintComponent()` method and put all your drawing operations in this method.

As you might have noticed, a `Graphics` object is sent to an interface component's `paintComponent()` method—not the `Graphics2D` you need. To create a `Graphics2D` object that represents the component's drawing surface, you must use casting to convert it, as in the following example:

Click here to view code image

```
public void paintComponent(Graphics comp) {
    Graphics2D comp2D = (Graphics2D) comp;
    // body of method
}
```

After a `comp2D` object has been cast from the `Graphics` object sent to the method as an argument, all drawing methods use this new object. The `Graphics` object will not be used again.

The Graphics Coordinate System

Java2D classes use the same (x, y) coordinate system you have used when setting the size of frames and other components in your Swing applications.

Java's coordinate system uses pixels as its unit of measure. The origin coordinate (0, 0) is in the upper-left corner of a component.

The value of x-coordinates increases to the right of (0, 0), and y-coordinates increase downward.

When you set a frame's size by calling its `setSize(int, int)` method, the frame's upper-left corner is at (0, 0), and its lower-right corner is at the

two arguments sent to `setSize()`.

The following statement creates a frame 425 pixels wide by 130 pixels tall with its lower-right corner at (425, 130):

```
setSize(425, 130);
```

Caution

Java2D differs from other drawing systems in which the (0, 0) origin is at the lower left and y values increase in an upward direction.

All pixel values are integers; you can't use decimal numbers to display something at a position between two integer values.

Figure 13.1 shows Java's graphical coordinate system visually, with the origin at (0, 0). Two of the points of a rectangle are at (20, 20) and (60, 60).

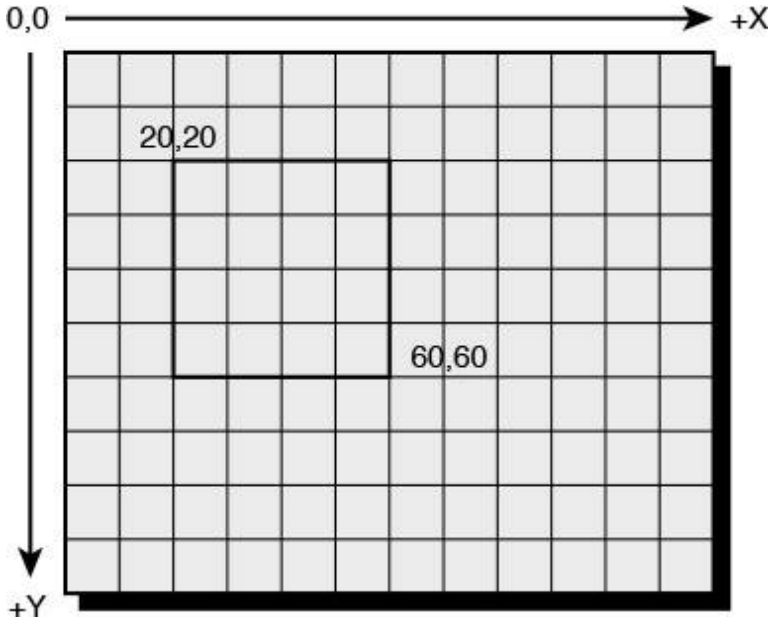


FIGURE 13.1 The Java graphics coordinate system.

Drawing Text

Text is the easiest thing to draw in Java2D. To draw text, call a `Graphics2D` object's `drawString(`*`String`*`, `*`int`*`, `*`int`*`)` method with three arguments:

- The string to display
- The x-coordinate where it should be displayed

- The y-coordinate where it should be displayed

The (x, y) coordinates used in the `drawString()` method represent the pixel at the lower-left corner of the string.

The following `paintComponent()` method draws the string "Free the bound periodicals" at the coordinates (22, 100):

Click here to view code image

```
public void paintComponent(Graphics comp) {
    Graphics2D comp2D = (Graphics2D) comp;
    comp2D.drawString("Free the bound periodicals", 22, 100);
}
```

This example uses a default font. To use a different font, you must create an object of the `Font` class in the `java.awt` package.

`Font` objects represent a font's name, style, and point size. A `Font` object is created by sending three arguments to its constructor:

- The font's name
- The font's style
- The font's point size

A font's name can be its physical name, such as Arial, Courier New, Garamond, or Turman Grotesk. If the font is present on the computer running the application, it is used. If the font is not present, the default font is used.

The name also can be one of five logical fonts: Dialog, DialogInput, Monospaced, SanSerif, or Serif. These fonts can be used to specify the kind of font to use without requiring a specific font. This often is a better choice, because some font families might not be present on all implementations of Java.

Three `Font` styles can be selected by using class variables: `PLAIN`, `BOLD`, and `ITALIC`. These constants are integers, and you can add them to combine effects.

The following statement creates a 24-point Dialog font that is bold and italicized:

Click here to view code image

```
Font f = new Font("Dialog", Font.BOLD + Font.ITALIC, 24);
```

After you have created a font, you can use it by calling the `setFont(`*`Font`*`)` method of the `Graphics2D` class with the font as the argument.

The `setFont()` method sets the current font, which will be used for all subsequent calls to the `drawString()` method on the same `Graphics2D` object until another font is set.

The following `paintComponent()` method creates a new `Font` object, sets the current font to that object, and draws the string "I'm deeply font of you" in 72-point type at the coordinates (13, 100):

Click here to view code image

```
public void paintComponent(Graphics comp) {
    Graphics2D comp2D = (Graphics2D) comp;
    Font f = new Font("Arial Narrow", Font.PLAIN, 72);
    comp2D.setFont(f);
    comp2D.drawString("I'm deeply font of you", 13, 100);
}
```

Java applications can ensure that a font is available by including it with the program and loading it from a file. This technique requires the `Font` class method `createFont(int, InputStream)`, which returns a `Font` object representing that font.

Input streams, which are covered on Day 15, "Working with Input and Output," are objects that can load data from a source such as a disk file or web address. The following statements load a font from a file named `Verdana.ttf` in the same folder as the class file that uses it:

Click here to view code image

```
try {
    File ttf = new File("Verdana.ttf");
    FileInputStream fis = new FileInputStream(ttf);
    Font font = Font.createFont(Font.TRUETYPE_FONT, fis);
} catch (IOException|FontFormatException exc) {
    System.out.println("Error: " + exc.getMessage());
}
```

The `try-catch` block handles input/output errors, which must be considered when data is loaded from a file. The `File`, `FileInputStream`, and `IOException` classes are part of the `java.io` package and are discussed in depth on Day 15.

When a font is loaded with `createFont()`, the `Font` object is 1 point and plain style. To change the size and style, call the font object's `deriveFont(int, int)` method with two arguments: the desired style and size.

Improving Fonts and Graphics with Antialiasing

If you displayed text using the skills introduced up to this point, the font's appearance would look crude compared to what you've come to expect from other software. Characters would be rendered with jagged edges, especially on curves and diagonal lines.

Java2D can draw fonts and graphics much more attractively using its support for *antialiasing*, a rendering technique that smooths out rough edges by altering the color of surrounding pixels.

This functionality is off by default. To turn it on, call a `Graphics2D` object's `setRenderingHint()` method with two arguments:

- A `RenderingHint.Key` object that identifies the rendering hint being set
- A `RenderingHint.Key` object that sets the value of that hint

The following code enables antialiasing on a `Graphics2D` object named `comp2D`:

Click here to view code image

```
comp2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
```

By calling this method in the `paintComponent()` method of a component, you can cause all subsequent drawing operations to employ antialiasing.

Finding Information About a Font

To make text look good in a graphical user interface, you often must figure out how much space the text is taking up on an interface component.

The `FontMetrics` class in the `java.awt` package provides methods to determine the size of the characters being displayed with a specified font, which can be used for things such as formatting and centering text.

The `FontMetrics` class can be used to find out detailed information about the current font, such as the width or height of characters it can display.

To use this class's methods, you must create a `FontMetrics` object using the `getFontMetrics()` method. The method takes a single argument: a `Font` object.

Table 13.1 shows some of the information you can find using font metrics. All these methods should be called on a `FontMetrics` object.

Method Name	Description
`stringWidth(String)`	Given a string, returns the full width of that string in pixels
`charWidth(char)`	Given a character, returns the width of that character
`getHeight()`	Returns the font's total height

TABLE 13.1 Font Metrics Methods

Listing 13.1 shows how the `Font` and `FontMetrics` classes can be used. The TextFrame application displays a string at the center of a frame, using font metrics to measure the string's width using the selected font. Create it in NetBeans in the `com.java21days` package.

LISTING 13.1 The Full Text of `TextFrame.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class TextFrame extends JFrame {
 8:     public TextFrame(String text, String fontName) {
 9:         super("Show Font");
10:         setSize(425, 150);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         TextFramePanel sf = new TextFramePanel(text,
fontName);
13:         add(sf);
14:         setVisible(true);
15:     }
16:
17:     public static void main(String[] arguments) {
18:         if (arguments.length < 1) {
19:             System.out.println("Usage: java TextFrame msg
font");
20:             System.exit(-1);
21:         }
22:         TextFrame tf = new TextFrame(arguments[0],
arguments[1]);
23:     }
24:
25: }
26:
27: class TextFramePanel extends JPanel {
```

```
28:     String text;
29:     String fontName;
30:
31:     public TextFramePanel(String text, String fontName) {
32:         super();
33:         this.text = text;
34:         this.fontName = fontName;
35:     }
36:
37:     public void paintComponent(Graphics comp) {
38:         super.paintComponent(comp);
39:         Graphics2D comp2D = (Graphics2D) comp;
40:         comp2D.setRenderingHint(RenderingHints.KEY_ANTIALIASI
NG,
41:             RenderingHints.VALUE_ANTIALIAS_ON);
42:         Font font = new Font(fontName, Font.BOLD, 18);
43:         FontMetrics metrics = getFontMetrics(font);
44:         comp2D.setFont(font);
45:         int x = (getSize().width - metrics.stringWidth(text))
/ 2;
46:         int y = getSize().height / 2;
47:         comp2D.drawString(text, x, y);
48:     }
49: }
```

The TextFrame application takes two command-line arguments, which you can set in NetBeans by choosing Project, Set Project Configuration, Customize. To run the application with this configuration, choose Run, Run Project.

Figure 13.2 shows how the application looks with a text message displayed in the font Arial Black. When you run the application, resize the frame window to see how the text moves so that it remains centered.

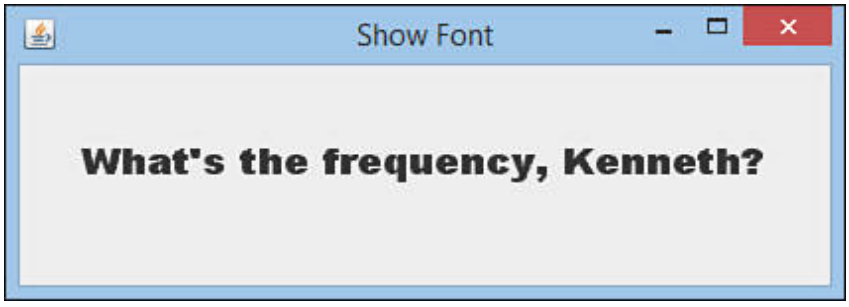


FIGURE 13.2 Displaying centered text in a graphical user interface.

The TextFrame application consists of two classes: a frame and a panel subclass called `TextFramePanel`. The text is drawn on the panel by overriding the `paintComponent(`*`Graphics`*`)` method and calling drawing methods of the `Graphics2D` class inside the method.

The `getSize()` method calls in lines 45 and 46 use the panel's width and height to determine where the text should be displayed. When the application is resized, the panel also is resized, and `paintComponent()` is called automatically.

Color

The `Color` class (in the `java.awt` package) and `ColorSpace` class (in `java.awt.color`) can be used to make a graphical user interface more, well, colorful. With these classes, you can set the color for use in drawing operations, as well as the background color of an interface component and other windows. You also can translate a color from one color system into another.

By default, Java uses colors according to the sRGB color system, which describes each shade by the amounts of red, green, and blue it contains (R, G, and B). Each of the three components can be represented as an integer between 0 and 255. Black is 0, 0, 0—the absence of any red, green, or blue. White is 255, 255, 255—the maximum amount of all three colors. You also can represent sRGB values using three floating-point numbers ranging from 0 to 1.0. Java can represent millions of colors between the two extremes using sRGB.

A color system is called a *color space*, and sRGB is only one such space. There also is XYZ, which was created by an international conference in 1931. Java supports the use of any color space desired as long as a `ColorSpace` object is used that defines the description system. You also can convert from any color space to sRGB, and vice versa.

Java's internal representation of colors using sRGB is just one color space used in a program. An output device such as a monitor or printer also has its own color space.

When you display or print something of a designated color, the output device might not support the designated color. In this circumstance, a different color is substituted or a dithering pattern is used to approximate the unavailable color.

The practical reality of color management is that the color you designate with sRGB will not be available on all output devices. If you need more precise control of the color, you can use `ColorSpace` and other classes in the `java.awt.color` package.

For most needs, the built-in use of sRGB to define colors should be sufficient.

Using `Color` Objects

Colors are represented by `Color` objects, which can be created with a constructor or by using one of the small number of standard colors available in the `Color` class.

You can call the `Color()` constructor to create a color with three integers that represent the sRGB value of the desired color or three floating-point numbers that serve the same purpose:

Click here to view code image

```
Color c1 = new Color(0.807F, 1F, 0F);

Color c2 = new Color(255, 204, 102);
```

The `c1` object describes a neon green color, and `c2` is butterscotch.

Note

It's easy to confuse floating-point literals such as 0F and 1F with hexadecimal numbers, which were discussed on Day 2, "The ABCs of Programming." Colors often are expressed in hexadecimal, such as when a background color is set on a web page using Cascading Style Sheets. The Java classes and methods you work with don't take hexadecimal arguments, so when you see a literal such as 1F or 0F, you're dealing with floating-point numbers.

Testing and Setting the Current Colors

The current color for drawing is designated by using the `setColor()` method of the `Graphics2D` class. This method must be called on the `Graphics2D` object that represents the area on which something is being drawn.

Several of the most common colors are available as class variables in the `Color` class. These colors use the following `Color` variables (sRGB values appear in parentheses):

Click here to view code image

```
black (0, 0, 0)                  magenta (255, 0, 255)
blue (0, 0, 255)                 orange (255, 200, 0)
```

```
cyan (0, 255, 255)              pink (255, 175, 175)
darkGray (64, 64, 64)           red (255, 0, 0)
gray (128, 128, 128)            white (255, 255, 255)
green (0, 255, 0)               yellow (255, 255, 0)
lightGray (192, 192, 192)
```

The following statement sets the color for a `Graphics2D` object named `comp2D` by using one of the standard class variables:

```
comp2D.setColor(Color.pink);
```

If you have created a `Color` object, it can be set in a similar fashion:

Click here to view code image

```
Color brush = new Color(255, 204, 102);
comp2D.setColor(brush);
```

After you set the current color, subsequent methods to draw strings and other graphics will use that color.

You can set the background color for a component, such as a panel or frame, by calling the component's `setBackground(`*`Color`*`)` method.

The `setBackground()` method sets the component's background color, as in this example:

```
setBackground(Color.white);
```

If you want to find out what the current color is, you can use the `getColor()` method on a `Graphics2D` object, or the `getBackground()` method on the component.

The following statement sets the current color of `comp2D`—a `Graphics2D` object—to the same color as a component's background:

Click here to view code image

```
comp2D.setColor(getBackground());
```

Drawing Lines and Polygons

All the basic drawing commands covered today are `Graphics2D` methods called within a component's `paintComponent()` method.

This is an ideal place for all drawing operations because `paintComponent()` is automatically called any time the component needs to be redisplayed.

If another program's window overlaps the component and it needs to be redrawn, putting all the drawing operations in `paintComponent()` ensures that no part of the drawing is left out.

Java2D features include the following:

- The capability to draw empty polygons and polygons filled with a solid color
- Special fill patterns, such as gradients and patterns
- Strokes that define the width and style of a drawing stroke
- Antialiasing to smooth edges of drawn objects

User and Device Coordinate Spaces

One concept introduced with Java2D is the difference between an output device's coordinate space and the coordinate space you refer to when drawing an object. Coordinate space is any 2D area that can be described using (x, y) coordinates.

For all drawing operations prior to Java, the only coordinate space used was the device coordinate space. You specified the (x, y) coordinates of an output surface, such as a panel, and those coordinates were used to draw text and other elements.

Java2D requires a second coordinate space that you refer to when creating an object and actually drawing it. This is called the *user coordinate space*.

Before any 2D drawing has occurred in a program, the device space and user space have the (0, 0) coordinates in the same place—the upper-left corner of the drawing area.

The user space's (0, 0) coordinates can move as a result of the 2D drawing operations being conducted. The x- and y-axes even can shift because of a 2D rotation. You'll learn more about the two coordinate systems as you work with Java2D.

Specifying the Rendering Attributes

The next step in 2D drawing is to specify how a drawn object is rendered. Java2D offers a wide range of attributes for designating color, including line width, fill patterns, transparency, and many other features.

Fill Patterns

Fill patterns control how a drawn object will be filled in. With Java2D, you can use a solid color, gradient fill, texture, or pattern of your own devising.

A fill pattern is defined by using the `setPaint(Paint)` method of `Graphics2D` with a `Paint` object as its only argument. Any class that can be a fill pattern, including `GradientPaint`, `TexturePaint`, and `Color`, can implement the `Paint` interface. Using a `Color` object with `setPaint()` is the same thing as using a solid color as the pattern.

A *gradient fill* is a gradual shift from one color at one coordinate point to another color at a different coordinate point. The shift can occur once between the points—which is called an *acyclic gradient*—or it can happen repeatedly, which is a *cyclic gradient*.

Figure 13.3 shows examples of acyclic and cyclic gradients between white and a darker color. The arrows indicate the points that the colors shift between.

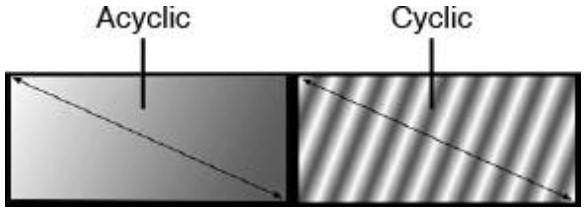


FIGURE 13.3 Acyclic and cyclic gradient shifts.

The coordinate points in a gradient do not refer directly to points on the `Graphics2D` object being drawn onto. Instead, they refer to user space and even can be outside the object being filled with a gradient.

Figure 13.4 illustrates this. Both rectangles are filled using the same `GradientPaint` object as a guide. One way to think of a gradient pattern is as a piece of fabric that has been spread over a flat surface. The shapes being filled with a gradient are the patterns cut from the fabric, and more than one pattern can be cut from the same piece of cloth.

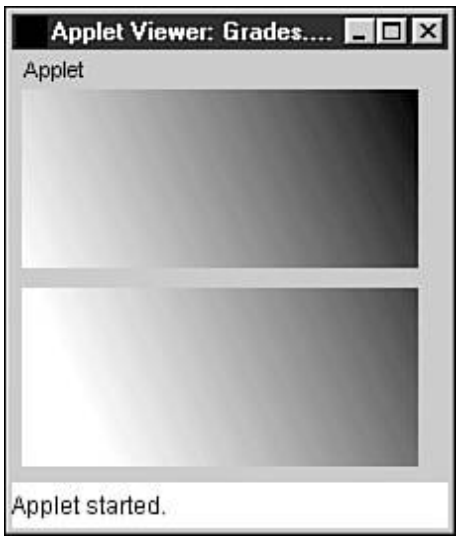


FIGURE 13.4 Two rectangles using the same `GradientPaint`.

A call to the `GradientPaint` constructor method takes the following format:

Click here to view code image

```
GradientPaint gp = new GradientPaint(
    x1, y1, color1, x2, y2, color2);
```

The point `(x1, y1)` is where the color represented by `color1` begins, and `(x2, y2)` is where the shift ends at `color2`.

If you want to use a cyclic gradient shift, an extra argument is added at the end:

Click here to view code image

```
GradientPaint gp = new GradientPaint(
    x1, y1, color1, x2, y2, color2, true);
```

The last argument is a Boolean value that is `true` for a cyclic shift. A `false` argument can be used for acyclic shifts, or you can omit this argument; acyclic shifts are the default behavior.

After you have created a `GradientPaint` object, set it as the current paint attribute by using the `setPaint()` method. The following statements create and select a gradient:

Click here to view code image

```
GradientPaint pat = new GradientPaint(0f, 0f, Color.white,
    100f, 45f, Color.blue);
comp2D.setPaint(pat);
```

All subsequent drawing operations to the `comp2D` object use this fill pattern until another one is chosen.

Setting a Drawing Stroke

Java2D allows you to vary the width of drawn lines by using the `setStroke()` method with a `BasicStroke`.

A simple `BasicStroke` constructor takes three arguments:

- A `float` value representing the line width, with `1.0` as the norm
- An `int` value determining the style of cap decoration drawn at the end of a line
- An `int` value determining the style of juncture between two line segments

The endcap- and juncture-style arguments use `BasicStroke` class variables. Endcap styles apply to the ends of lines that do not connect to other lines. Juncture styles apply to the ends of lines that join other lines.

Possible endcap styles are `CAP_BUTT` for no endpoints, `CAP_ROUND` for circles around each endpoint, and `CAP_SQUARE` for squares. Figure 13.5 shows each endcap style. As you can see, the only visible difference between the `CAP_BUTT` and `CAP_SQUARE` styles is that `CAP_SQUARE` is longer because of the added square endcap.

FIGURE 13.5 Endpoint cap styles.

Possible juncture styles include `JOIN_MITER`, which joins segments by extending their outer edges, `JOIN_ROUND`, which rounds off a corner between two segments, and `JOIN_BEVEL`, which joins segments with a straight line. Figure 13.6 shows examples of each juncture style.

FIGURE 13.6 Endpoint juncture styles.

The following statements create a `BasicStroke` object and make it the current stroke:

Click here to view code image

```
BasicStroke pen = new BasicStroke(2.0F,
    BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_ROUND);
comp2D.setStroke(pen);
```

The stroke has a width of 2 pixels, plain endpoints, and rounded segment corners.

Creating Objects to Draw

After you have created a `Graphics2D` object and specified the rendering attributes, the final two steps are to create the object and draw it.

You create a drawn object in Java2D by defining it as a geometric shape using a class in the `java.awt.geom` package. You can draw lines, rectangles, ellipses, arcs, and polygons.

The `Graphics2D` class does not have a different method for each shape you can draw. Instead, you define the shape and use it as an argument to `draw()` or `fill()` methods.

Lines

Lines are created using the `Line2D.Float` class. This class takes four arguments: the (x,y) coordinates of one endpoint followed by the (x,y) coordinates of the other. Here's an example:

Click here to view code image

```
Line2D.Float ln = new Line2D.Float(60F, 5F, 13F, 28F);
```

This statement creates a line between (60, 5) and (13, 28). Note that an F is used with the literals sent as arguments. Otherwise, the Java compiler would assume that the values were integers.

Rectangles

Rectangles are created by using the `Rectangle2D.Float` class or `Rectangle2D.Double` class. The difference between the two is that one takes `float` arguments, and the other takes `double` arguments.

`Rectangle2D.Float` takes four arguments: x-coordinate, y-coordinate, width, and height. The following is an example:

Click here to view code image

```
Rectangle2D.Float rc = new Rectangle2D.Float(10F, 13F, 40F, 20F);
```

This creates a rectangle at 10, 13 that is 40 pixels wide by 20 pixels tall.

Ellipses

Ellipses can be created with the `Ellipse2D.Float` class. It takes four arguments: x-coordinate, y-coordinate, width, and height.

The following statement creates an ellipse at (113, 25) with a width of 22 pixels and a height of 40 pixels:

Click here to view code image

```
Ellipse2D.Float ee = new Ellipse2D.Float(113, 25, 22, 40);
```

Arcs

Of all the shapes you can draw in Java2D, arcs are the most complex to construct.

Arcs are created with the `Arc2D.Float` class, which takes seven arguments:

- The (x,y) coordinates of an invisible ellipse that would include the arc if it were drawn (first two arguments)
- The width and height of the ellipse (third and fourth arguments)
- The starting degree of the arc
- The number of degrees it travels on the ellipse
- An integer describing how the arc is closed

The number of degrees the arc travels is specified in a counterclockwise direction by using negative numbers.

Figure 13.7 shows where degree values are located when determining an arc's starting degree. The arc's starting angle ranges from 0 to 359 degrees counterclockwise. On a circular ellipse, 0 degrees is at the 3 o'clock position, 90 degrees is at 12 o'clock, 180 degrees is at 9 o'clock, and 270 degrees is at 6 o'clock.

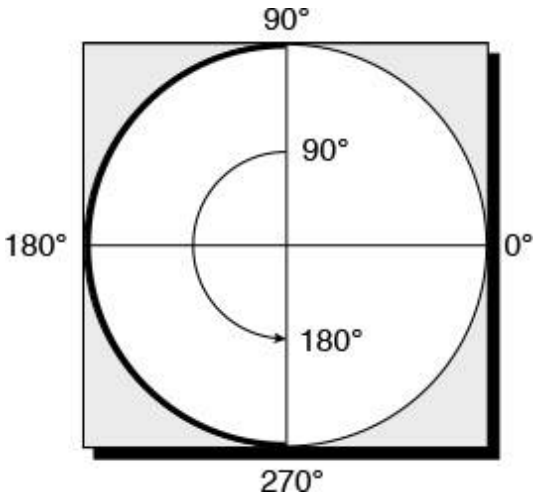


FIGURE 13.7 Determining the starting degree of an arc.

The last argument to the `Arc2D.Float` constructor uses one of three class variables: `Arc2D.OPEN` for an unclosed arc, `Arc2D.CHORD` to connect the arc's endpoints with a straight line, and `Arc2D.PIE` to connect the arc to the center of the ellipses like a pie slice. Figure 13.8 shows each of these styles.

FIGURE 13.8 Arc closure styles.

Note

The `Arc2D.OPEN` closure style does not apply to filled arcs. A filled arc that has `Arc2D.OPEN` as its style will be closed using the same style as `Arc2D.CHORD`.

The following statement creates an `Arc2D.Float` object:

Click here to view code image

```
Arc2D.Float arc = new Arc2D.Float(
    27F, 22F, 42F, 30F, 33F, 90F, Arc2D.PIE);
```

This creates an arc for an oval at (27, 22) that is 42 pixels wide by 30 pixels tall. The arc begins at 33 degrees, extends 90 degrees clockwise, and is closed like a pie slice.

Polygons

You create polygons in Java2D by defining each movement from one point on the polygon to another. A polygon can be formed from straight lines, quadratic curves, or Bézier curves.

The movements to create a polygon are defined as a `GeneralPath` object, which also is part of the `java.awt.geom` package.

A `GeneralPath` object can be created without any arguments, as shown here:

Click here to view code image

```
GeneralPath polly = new GeneralPath();
```

The `moveTo()` method of `GeneralPath` is used to create the first point on the polygon. The following statement would be used if you wanted to start `polly` at the coordinate 5, 0:

```
polly.moveTo(5F, 0F);
```

After creating the first point, the `lineTo()` method is used to create lines that end at a new point. This method takes two arguments: the (x,y) coordinates of the new point.

The following statements add three lines to the `polly` object:

```
polly.lineTo(205F, 0F);
polly.lineTo(205F, 90F);
polly.lineTo(5F, 90F);
```

The `lineTo()` and `moveTo()` methods require `float` arguments to specify coordinate points.

If you want to close a polygon, the `closePath()` method is used without any arguments, as shown here:

```
polly.closePath();
```

This method closes a polygon by connecting the current point with the point specified by the most recent `moveTo()` method. You can close a polygon without this method by using a `lineTo()` method that connects to the original point.

After you have created an open or closed polygon, you can draw it like any other shape using the `draw()` and `fill()` methods. The `polly` object is a rectangle with points at (5, 0), (205, 0), (205, 90), and (5, 90).

Drawing Objects

After you have defined the rendering attributes, such as color and line width, and have created the object to be drawn, you're ready to draw something in all its 2D glory.

All drawn objects use the same `Graphics2D` class's methods: `draw()` for outlines and `fill()` for filled objects. These take an object as the only argument.

Drawing a Map

The next project you will create is an application that draws a simple map using 2D drawing techniques. Create the `Map` class in the `com.java21days` package in NetBeans and fill it with the code in Listing 13.2.

LISTING 13.2 The Full Text of `Map.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.geom.*;
 5: import javax.swing.*;
 6:
 7: public class Map extends JFrame {
 8:     public Map() {
 9:         super("Map");
10:         setSize(360, 350);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         MapPane map = new MapPane();
13:         add(map);
14:         setVisible(true);
15:     }
16:
17:     public static void main(String[] arguments) {
18:         Map frame = new Map();
19:     }
20:
21: }
22:
23: class MapPane extends JPanel {
24:     public void paintComponent(Graphics comp) {
25:         Graphics2D comp2D = (Graphics2D) comp;
26:         comp2D.setColor(Color.blue);
```

```
27:          comp2D.setRenderingHint(RenderingHints.KEY_ANTIALIASI
NG,
28:              RenderingHints.VALUE_ANTIALIAS_ON);
29:          Rectangle2D.Float background = new Rectangle2D.Float(
30:              0F, 0F, getSize().width, getSize().height);
31:          comp2D.fill(background);
32:          // Draw waves
33:          comp2D.setColor(Color.white);
34:          BasicStroke pen = new BasicStroke(2F,
35:          BasicStroke.CAP_BUTT, BasicStroke.JOIN_ROUND);
36:          comp2D.setStroke(pen);
37:          for (int ax = 0; ax < 340; ax += 10) {
38:              for (int ay = 0; ay < 340 ; ay += 10) {
39:                  Arc2D.Float wave = new Arc2D.Float(ax, ay,
40:                      10, 10, 0, -180, Arc2D.OPEN);
41:                  comp2D.draw(wave);
42:              }
43:          }
44:          // Draw Florida
45:          GradientPaint gp = new GradientPaint(0F, 0F,
Color.green,
46:              350F,350F, Color.orange, true);
47:          comp2D.setPaint(gp);
48:          GeneralPath fl = new GeneralPath();
49:          fl.moveTo(10F, 12F);
50:          fl.lineTo(234F, 15F);
51:          fl.lineTo(253F, 25F);
52:          fl.lineTo(261F, 71F);
53:          fl.lineTo(344F, 209F);
54:          fl.lineTo(336F, 278F);
55:          fl.lineTo(295F, 310F);
56:          fl.lineTo(259F, 274F);
57:          fl.lineTo(205F, 188F);
58:          fl.lineTo(211F, 171F);
59:          fl.lineTo(195F, 174F);
60:          fl.lineTo(191F, 118F);
61:          fl.lineTo(120F, 56F);
62:          fl.lineTo(94F, 68F);
63:          fl.lineTo(81F, 49F);
64:          fl.lineTo(12F, 37F);
65:          fl.closePath();
66:          comp2D.fill(fl);
67:          // Draw ovals
68:          comp2D.setColor(Color.black);
69:          BasicStroke pen2 = new BasicStroke();
70:          comp2D.setStroke(pen2);
71:          Ellipse2D.Float e1 = new Ellipse2D.Float(235, 140,
15, 15);
72:          Ellipse2D.Float e2 = new Ellipse2D.Float(225, 130,
15, 15);
73:           Ellipse2D.Float e3 = new Ellipse2D.Float(245, 130,
```

```
15, 15);
74:             comp2D.fill(e1);
75:             comp2D.fill(e2);
76:             comp2D.fill(e3);
77:         }
78: }
```

In the Map application, line 4 imports the classes in the `java.awt.geom` package. This statement is required because `import java.awt.*;` in line 1 handles only classes, not packages, available under `java.awt`.

Line 25 creates the `comp2D` object used for all 2D drawing operations. It's a cast of the `Graphics` object that represents the panel's visible surface.

Lines 34–36 create a `BasicStroke` object that represents a line width of 2 pixels and then makes this the current stroke with the `setStroke()` method of `Graphics2D`.

Lines 37–42 use two nested `for` loops to create waves from individual arcs.

Lines 45–46 create a gradient fill pattern from the color green at (0, 0) to orange at (50, 50). The last argument to the constructor, `true`, causes the fill pattern to repeat itself as many times as needed to fill an object.

Line 47 sets the current gradient fill pattern using the `setPaint()` method and the `gp` object just created.

Lines 48–66 create the polygon shaped like the author's home state and draw it. This polygon is filled with a green-to-orange gradient pattern.

Line 68 sets the current color to black. This replaces the gradient fill pattern for the next drawing operation because colors are also fill patterns.

Line 69 creates a new `BasicStroke()` object with no arguments, which defaults to a 1-pixel line width.

Line 70 sets the current line width to the new `BasicStroke` object `pen2`.

Lines 71–73 create three ellipses at (235, 140), (225, 130), and (245, 130). Each is 15 pixels wide by 15 pixels tall, making them circles.

Figure 13.9 shows the application running.

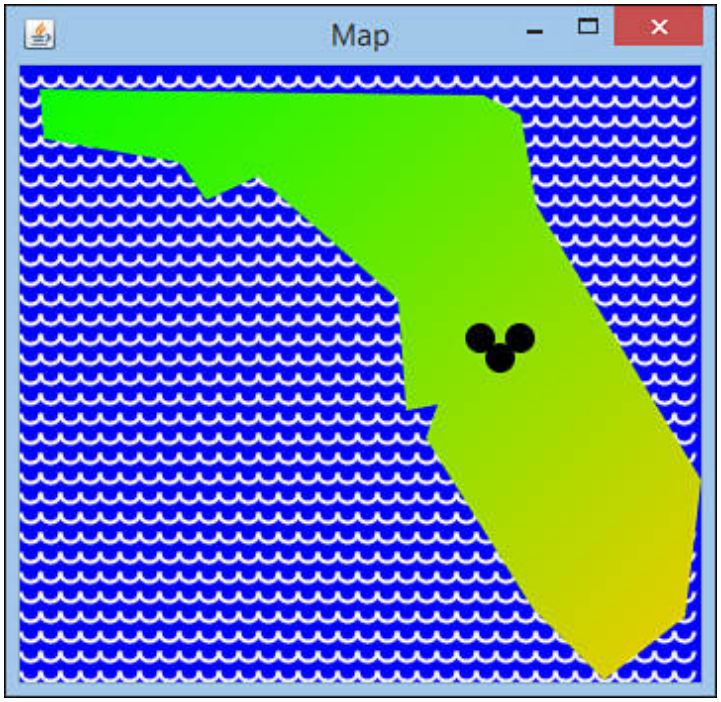


FIGURE 13.9 The Map application.

Summary

You now have some tools to improve the looks of a Java program. You can draw with lines, rectangles, ellipses, polygons, fonts, colors, and patterns onto a frame, a panel, and other user interface components using Java2D.

Java2D uses the same two methods for each drawing operation—`draw()` and `fill()`. Different objects are created using classes of the `java.awt.geom` package, and these are used as arguments for the drawing methods of `Graphics2D`.

Tomorrow, on Day 14, "Developing Swing Applications," you'll learn how to create applications that are launched from a web page using Java Web Start technology.

Q&A

Q What does the uppercase F refer to in source code today? It is added to coordinates, as in the method `polly.moveTo(5F, 0F)`. Why is F used for these coordinates and not others, and why is a lowercase f used elsewhere?

A The F or f indicates that a number is a floating-point number rather than an integer, and uppercase and lowercase can be used interchangeably. If

you don't use one of them, the Java compiler assumes that the number is an `int` value. Many methods and constructors in Java require floating-point arguments but can handle integers because an integer can be converted to floating-point without changing its value. For this reason, constructors such as `Arc2D.Float()` can use arguments such as `10` and `180` instead of `10F` and `180F`.

Q The section "Improving Fonts and Graphics with Antialiasing" mentioned a class called `RenderingHint.Key`. Why does this class have two names separated by a period? What does this signify?

A The use of two names to identify a class indicates that it is an inner class. The first class name is the enclosing class, followed by a period and the name of the inner class. In this case, the `Key` class is an inner class within the `RenderingHint` class.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What object is required before you can draw something in Java using Swing?

   A. `Graphics2D`

   B. `WindowListener`

   C. `JFrame`

2. Which of the following is not a valid Java statement to create a `Color` object?

   A. `Color c1 = new Color(0F, 0F, 0F);`

   B. `Color c2 = new Color(0, 0, 0);`

   C. Both are valid.

3. What does `getSize().width` refer to?

   A. The width of the interface component's window

   B. The width of the frame's window

   C. The width of any graphical user interface component in Java

Answers

1. A. The `Graphics2D` object is cast from a `Graphics` object and represents a graphics context for a graphical user interface component.
2. C. Both are valid ways to create the object. You also can use hexadecimal values to create a `Color`, as in this example:

Click here to view code image

```
Color c3 = new Color(0xFF, 0xCC, 0x66);
```

3. C. You can call `getSize().width` and `getSize().height` on any user interface component.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
import java.awt.*;
import javax.swing.*;

public class Result extends JFrame {
    public Result() {
        super("Result");
        JLabel width = new JLabel("This frame is " +
            getSize().width + " pixels wide.");
        add("North", width);
        setSize(220, 120);
    }

    public static void main(String[] arguments) {
        Result r = new Result();
        r.setVisible(true);
    }
}
```

What will be the reported width of the frame, in pixels, when the application runs?

A. 0 pixels

B. 120 pixels

C. 220 pixels

D. The width of the user's monitor

The answer is available on the book's website at www.java21days.com. Visit the Day 13 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create an application that draws a circle, with its radius, (x,y) position, and color all determined by arguments.
2. Create an application that draws a pie graph.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 14. Developing Swing Applications

The first exposure many people had to the Java programming language was applets—small, security-restricted Java programs that run on web pages. Java Web Start, a protocol for downloading and running Java programs, makes it possible to launch applications from a web page as if they were applets.

Today, you learn how to create these web-launched Java programs as you explore the following topics:

- How to install and run Java applications in a web browser
- How to publish your application's files and deploy it
- How Swing applications can run into performance slowdowns on time-consuming tasks
- How to address these problems using `SwingWorker`, a class that performs Swing work in its own thread

Java Web Start

One of the issues you must deal with as a Java programmer is how to make your software available to your users.

Java applications require a Java Virtual Machine (JVM), so one must be included with the application, previously installed on a computer, or installed by users. The easiest solution (for you) is to require that users download and install the Java Runtime Environment from Oracle's website at www.java.com.

Regardless of how you deal with the requirement for a JVM, you distribute an application like any other program—making it available for download, distributing it on a CD, or using some other means. A user must run an installation program to set it up, if one is available, or copy the files and folders manually.

Java eases the challenges of software deployment with Java Web Start, a way to run Java applications presented on a web page and stored on a web server. Here's how it works:

1. A programmer packages an application and all the files it needs into a JAR archive, along with a file that uses the Java Network Launching Protocol (JNLP), part of Java Web Start.

2. The file is stored on a web server with a web page that links to that file.
3. A user loads the page with a browser and clicks the link.
4. If the user does not have the Java Runtime Environment, a dialog box opens, asking whether the JRE should be downloaded and installed. The full installation is from 40 to 70MB in size (depending on operating system).
5. The Java Runtime Environment installs and runs the program, opening new frames and other interface components like any other application. The program is saved in a cache, so it can be run again later without requiring installation.

To see Java Web Start in action, visit Oracle's Java Web Start site at www.oracle.com/technetwork/java/javase/javawebstart. Click the Code Samples & Apps link, and then the Demos link. The Web Start Demos page, shown in Figure 14.1, contains pictures of several Java applications, each with a Click to Launch! button you can use to run the application.

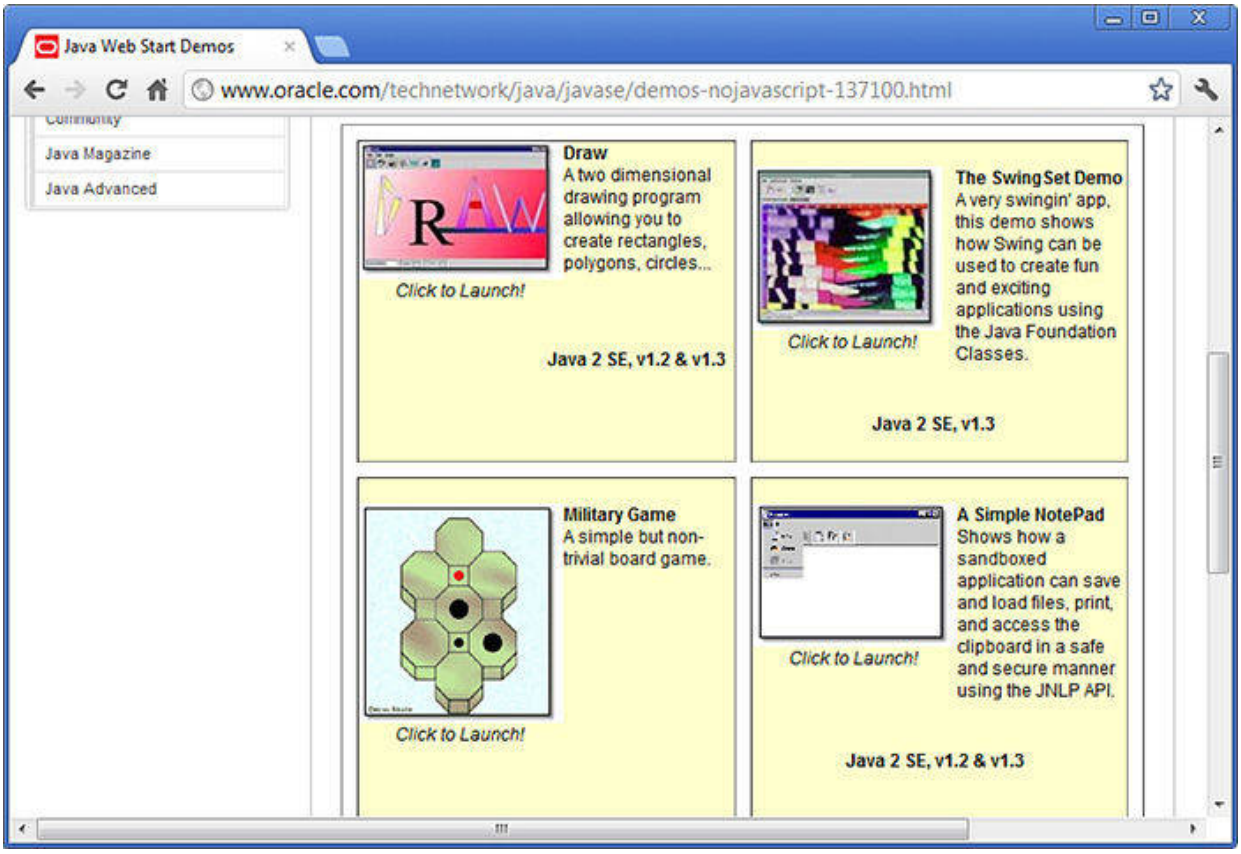


FIGURE 14.1 Presenting Web Start applications on a web page.

Click the Click to Launch! button of one of the applications. If you don't have the Java Runtime Environment yet, a dialog box opens, asking whether you want to download and install it.

The runtime environment includes the Java Plug-in, a JVM that adds support for the current version of the language to browsers. The environment also can be used to run applications, regardless of whether they use Java Web Start.

When an application is run using Java Web Start, a title screen appears briefly, and then the application's graphical user interface appears.

Note

If you have installed NetBeans or the JDK, you probably have the Java Runtime Environment on your computer already.

Figure 14.2 shows one of the demo applications that Oracle offers, a military strategy game in which three black dots attempt to keep a red dot from moving into their territory.

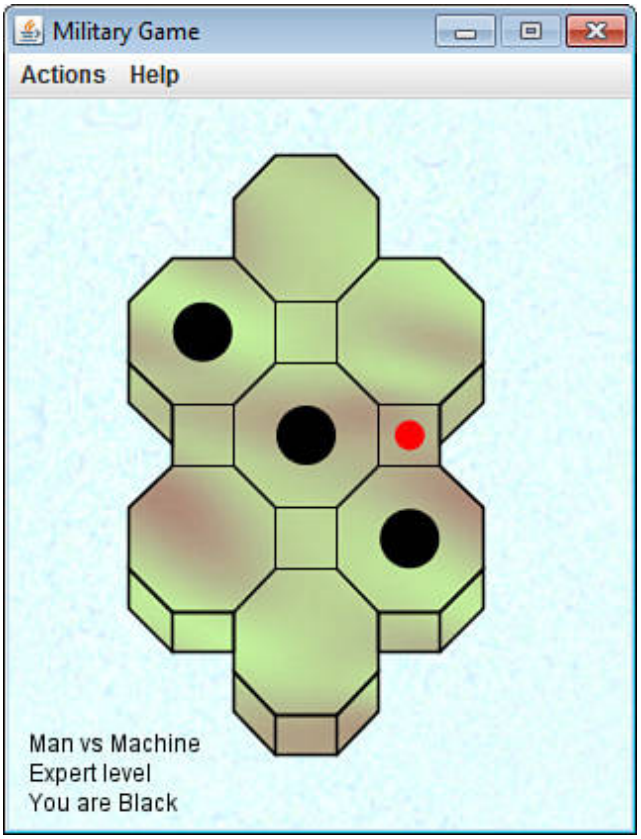


FIGURE 14.2 Running a Java Web Start application.

As you can see in Figure 14.2, the application looks no different from any other application. Unlike applets, which are presented in conjunction with a web page, applications launched with Java Web Start run in their own windows, as if they were run from a command line.

One thing that's different about a Java Web Start application is the security that can be offered to users. When an application attempts to do something, such as read or write files, the user can be asked for permission.

For example, another of the demo programs is a text editor. When you try to save a file for the first time with this application, the Security Warning dialog box opens, as shown in Figure 14.3.

FIGURE 14.3 Choosing an application's security privileges.

If the user does not permit something that requires such authorization, the application cannot function fully. The kinds of things that trigger a security dialog box are reading and writing files, loading network resources from servers other than the one hosting the program, and the like.

After Java Web Start has run an application, it is stored on a user's computer in a cache, enabling it to be run again later without installation. The only exception is when a new version of the application becomes available. In this case, the new version is downloaded and installed automatically in place of the existing one.

Note

Although you run a Java Web Start application for the first time using a web browser, that's not a requirement. The applications can be run from a desktop shortcut.

The default security restrictions in place for a Java Web Start application can be overridden if it is stored in a digitally signed Java archive. The user is presented with the signed security certificate, which documents the program's author and the certificate-granting authority vouching for its identity, and is asked whether to accept or reject it. The application won't run unless the certificate has been accepted.

Using Java Web Start

Any Java application can be run using Java Web Start as long as the web server that offers the application is configured to work with the technology and all the class files and other files it needs have been packaged together.

To prepare an application to use Java Web Start, you must save the application's files in a JAR file (Java archive), create a special Java Web Start configuration file for the application, and upload the files to the web server.

The configuration file that must be created uses JNLP, an Extensible Markup Language (XML) file format that specifies the application's main class file, its JAR archive, and other things about the program.

Note

XML is introduced during Day 20, "XML Web Services." Because the format of JNLP files is relatively self-explanatory, you don't need to know much about XML to create a JNLP file.

The next project you will undertake is using Java Web Start to launch and run PageData, an application that displays information about web pages.

Creating a JNLP File

The first thing you must do is package all of an application's class files into a JAR file, along with any other files it needs. NetBeans creates a JAR file automatically for each project you build in the IDE.

Because you've been using one project for all the projects in the preceding 13 days, a new project is needed:

1. Choose File, New Project. The New Project dialog appears.

2. Choose Java in the Categories pane and Java Applications in the Projects pane, and then click Next. The New Java Application dialog opens.
3. Enter `PageData` as the Project Name.
4. Select the Create Main Class check box.
5. Enter `PageData` in the text field next to Create Main Class.
6. Click Finish.

The file `PageData.java` opens in NetBeans' source code editor with some starter code entered for you. Delete all this code and enter the code shown in Listing 14.1 as the `PageData` class in the `com.java21days` package.

LISTING 14.1 The Full Text of `PageData.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import java.net.*;
 6: import java.io.*;
 7: import javax.swing.*;
 8:
 9: public class PageData extends JFrame implements
ActionListener,
10:     Runnable {
11:
12:     Thread runner;
13:     String[] headers = { "Content-Length", "Content-Type",
14:         "Date", "Public", "Expires", "Last-Modified",
15:         "Server" };
16:
17:     URL page;
18:     JTextField url;
19:     JLabel[] headerLabel = new JLabel[7];
20:     JTextField[] header = new JTextField[7];
21:     JButton readPage, clearPage, quitLoading;
22:     JLabel status;
23:
24:     public PageData() {
25:         super("Page Data");
26:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27:         setLookAndFeel();
```

```
 28:         setLayout(new GridLayout(10, 1));
 29:
 30:         JPanel first = new JPanel();
 31:         first.setLayout(new FlowLayout(FlowLayout.RIGHT));
 32:         JLabel urlLabel = new JLabel("URL:");
 33:         url = new JTextField(22);
 34:         urlLabel.setLabelFor(url);
 35:         first.add(urlLabel);
 36:         first.add(url);
 37:         add(first);
 38:
 39:         JPanel second = new JPanel();
 40:         second.setLayout(new FlowLayout());
 41:         readPage = new JButton("Read Page");
 42:         clearPage = new JButton("Clear Fields");
 43:         quitLoading = new JButton("Quit Loading");
 44:         readPage.setMnemonic('r');
 45:         clearPage.setMnemonic('c');
 46:         quitLoading.setMnemonic('q');
 47:         readPage.setToolTipText("Begin Loading the Web
Page");
 48:         clearPage.setToolTipText("Clear All Header Fields
Below");
 49:         quitLoading.setToolTipText("Quit Loading the Web
Page");
 50:         readPage.setEnabled(true);
 51:         clearPage.setEnabled(false);
 52:         quitLoading.setEnabled(false);
 53:         readPage.addActionListener(this);
 54:         clearPage.addActionListener(this);
 55:         quitLoading.addActionListener(this);
 56:         second.add(readPage);
 57:         second.add(clearPage);
 58:         second.add(quitLoading);
 59:         add(second);
 60:
 61:         JPanel[] row = new JPanel[7];
 62:         for (int i = 0; i < 7; i++) {
 63:             row[i] = new JPanel();
 64:             row[i].setLayout(new
FlowLayout(FlowLayout.RIGHT));
 65:             headerLabel[i] = new JLabel(headers[i] + ":");
 66:             header[i] = new JTextField(22);
 67:             headerLabel[i].setLabelFor(header[i]);
 68:             row[i].add(headerLabel[i]);
 69:             row[i].add(header[i]);
 70:             add(row[i]);
 71:         }
 72:
 73:         JPanel last = new JPanel();
 74:         last.setLayout(new FlowLayout(FlowLayout.LEFT));
```

```
 75:          status = new JLabel("Enter a URL address to
check.");
 76:          last.add(status);
 77:          add(last);
 78:          pack();
 79:          setVisible(true);
 80:      }
 81:
 82:      public void actionPerformed(ActionEvent evt) {
 83:          Object source = evt.getSource();
 84:          if (source == readPage) {
 85:              try {
 86:                  page = new URL(url.getText());
 87:                  if (runner == null) {
 88:                      runner = new Thread(this);
 89:                      runner.start();
 90:                  }
 91:                  quitLoading.setEnabled(true);
 92:                  readPage.setEnabled(false);
 93:              }
 94:              catch (MalformedURLException e) {
 95:                  status.setText("Bad URL: " + page);
 96:              }
 97:          } else if (source == clearPage) {
 98:              for (int i = 0; i < 7; i++)
 99:                  header[i].setText("");
100:              quitLoading.setEnabled(false);
101:              readPage.setEnabled(true);
102:              clearPage.setEnabled(false);
103:          } else if (source == quitLoading) {
104:              runner = null;
105:              url.setText("");
106:              quitLoading.setEnabled(false);
107:              readPage.setEnabled(true);
108:              clearPage.setEnabled(false);
109:          }
110:      }
111:
112:      public void run() {
113:          URLConnection conn;
114:          try {
115:              conn = this.page.openConnection();
116:              conn.connect();
117:              status.setText("Connection opened ...");
118:              for (int i = 0; i < 7; i++)
119:                  header[i].setText(conn.getHeaderField(header
s[i]));
120:              quitLoading.setEnabled(false);
121:              clearPage.setEnabled(true);
122:              status.setText("Done");
123:              runner = null;
```

```
124:             }
125:             catch (IOException e) {
126:                 status.setText("IO Error:" + e.getMessage());
127:             }
128:         }
129:
130:         private static void setLookAndFeel() {
131:             try {
132:                 UIManager.setLookAndFeel(
133:                     "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
134:                 );
135:             } catch (Exception exc) {
136:                 // ignore error
137:             }
138:         }
139:
140:
141:         public static void main(String[] arguments) {
142:             PageData frame = new PageData();
143:         }
144: }
```

After you've saved the project, build it in NetBeans by choosing Run, Clean and Build Project. This extra step is required because you will deploy this application on the Web instead of simply running it on your computer.

The PageData application takes a web address (URL) as input and loads data associated with the page at that address. This program uses some networking techniques that will be explored fully during Day 17, "Communicating Across the Internet."

Next, the application needs an icon that will be displayed when it is loaded and used in menus and desktops. The icon for a Java Web Start application can be in either GIF or JPEG format and should be 64 pixels wide by 64 pixels tall.

For this project, if you don't want to create a new icon, you can download `pagedataicon.gif` from the book's website. Go to www.java21days.com and open the Day 14 page. Right-click the `pagedataicon.gif` link and save the file to a folder on your computer.

Next, click the Files tab to bring that pane to the front in NetBeans, as shown in Figure 14.4.

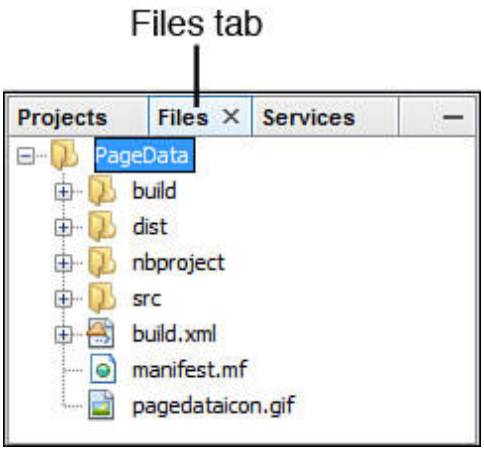


FIGURE 14.4 Adding a file to a project in NetBeans.

Scroll down to the PageData folder icon, and drag `pagedataicon.gif` from that folder into NetBeans. The file appears in the Files listing, as shown in Figure 14.4.

The final thing you must do is create the JNLP file that describes the application. Listing 14.2 is a JNLP file used to distribute the PageData application.

You can create this file in NetBeans:

1. Choose File, New File. The New File dialog opens.
2. Choose Other in the Categories pane.
3. Choose JNLP File in the File Types pane.
4. Click Next. The New JNLP File dialog appears.
5. Enter `PageData` in the File Name field. In the Created File field, you see that NetBeans automatically adds the file extension `.jnlp` to the filename.
6. Click Finish.

NetBeans creates a new JNLP file in the source code editor, starting you with some text in XML format. Delete all this text and enter the code shown in Listing 14.2. Then save the file.

LISTING 14.2 The Full Text of `PageData.jnlp`

Click here to view code image

```
1: <?xml version="1.0" encoding="utf-8"?>
2: <!-- JNLP File for the PageData Application -->
```

```
 3: <jnlp
 4:   codebase="http://cadenhead.org/book/java-21-days/java"
 5:   href="PageData.jnlp">
 6:   <information>
 7:     <title>PageData Application</title>
 8:     <vendor>Rogers Cadenhead</vendor>
 9:     <homepage href="http://www.java21days.com"/>
10:     <icon href="pagedataicon.gif"/>
11:     <offline-allowed/>
12:   </information>
13:   <resources>
14:     <j2se version="1.8"/>
15:     <jar href="PageData.jar"/>
16:   </resources>
17:   <security>
18:     <j2ee-application-client-permissions/>
19:   </security>
20:   <application-desc main-class="PageData"/>
21: </jnlp>
```

Because a JNLP file is structured as XML data, everything within the < and > symbols is a tag. Tags are placed around the information that the tag describes. There's an opening tag before the information and a closing tag after it.

For example, line 7 of Listing 14.2 contains the following text:

Click here to view code image

```
<title>PageData Application</title>
```

In order from left to right, this line contains the opening tag `<title>`, the text `PageData Application`, and the closing tag `</title>`. The text between the tags, "PageData Application," is the application's title. Java Web Start will display the title as the application is being loaded. The title also will be used in menus and shortcuts.

The difference between opening tags and closing tags is that closing tags begin with a slash character (/), and opening tags do not. In line 8, `<vendor>` is the opening tag, `</vendor>` is the closing tag, and these tags surround the name of the vendor who created the application. I've used my name here. Delete it and replace it with your own name, taking care not to alter the `<vendor>` or `</vendor>` tags around it.

Some tags have an opening tag only, such as line 11:

```
<offline-allowed/>
```

The `offline-allowed` tag indicates that the application can be run even if the user is not connected to the Internet. If it were omitted from the JNLP file, the opposite would be true, and the user would be forced to go online before running this application.

In XML, all tags that do not have a closing tag end with /> instead of >.

Tags also can have attributes, which are another way to define information in an XML file. An attribute is a name inside a tag that is followed by an equal sign and some text within quotes.

For example, consider line 9 of Listing 14.2:

Click here to view code image

```
<homepage href="http://www.java21days.com"/>
```

This is the `homepage` tag, and it has one attribute, `href`. The text between the quote marks is used to set the value of this attribute to "http://www.java21days.com". This defines the application's home page—the web page that users should visit if they want to read more about the program and how it works.

The PageData JNLP file defines a simple Java Web Start application that runs with security restrictions, as defined in lines 17–19:

Click here to view code image

```
<security>
  <j2ee-application-client-permissions/>
</security>
```

In addition to the tags that have already been described, Listing 14.2 defines other information required by Java Web Start.

Line 1 specifies that the file uses XML and the UTF-8 character set. This same line can be used on any of the JNLP files you create for applications.

Line 2 is a comment. Like comments in Java classes, this text is provided solely for the benefit of humans looking at this file. Java Web Start ignores it.

The `jnlp` element, which begins on line 3 and ends on line 21, must surround all the other tags that configure Web Start.

This tag has two attributes, `codebase` and `href`, which indicate where the JNLP file for this application can be found. The `codebase` attribute is the uniform resource locator (URL) of the folder that contains the JNLP file. The

`href` attribute is the name of the file or a relative URL that includes a folder and the name (such as `"pub/PageData.jnlp"`).

In Listing 14.2, the attributes indicate that the application's JNLP file is at the following web address:

Click here to view code image

```
http://cadenhead.org/book/java-21-days/java/PageData.jnlp
```

The `information` element (lines 6–12) defines information about the application. Elements can contain other elements in XML, and in Listing 14.2, the `information` element contains `title`, `vendor`, `homepage`, `icon`, and `offline-allowed` tags.

The `title`, `vendor`, `homepage`, and `offline-allowed` elements were described earlier.

The `icon` element (line 10) contains an `href` attribute that indicates the name (or folder location and name) of the program's icon. Like all file references in a JNLP file, this element uses the `codebase` attribute to determine the full URL of the resource. In this example, the `icon` element's `href` attribute is `pagedataicon.gif`, and the `codebase` is `"http://cadenhead.org/book/java21days/java"`, so the icon file is at the following web address:

Click here to view code image

```
http://cadenhead.org/book/java21days/java/pagedataicon.gif
```

The `resources` element (lines 13–16) defines resources used by the application when it runs.

The `j2se` element has a `version` attribute that indicates which version of the JVM should run the application. This attribute can specify a general version (such as `"1.7"` or `"1.8"`), a specific version (such as `"1.8.0-ea"`), or a reference to multiple versions. A general version number can be followed by a plus sign. The tag `<j2se version="1.6+">` sets up an application to be run by any JVM from version 1.6 upward.

Note

When you use the `j2se` element to specify multiple versions, Java Web Start does not use a beta version to run an application. The only way to run an application with a beta release is to indicate that release specifically.

The `jar` element has an `href` attribute that specifies the application's JAR file. This attribute can be a filename or a reference to a folder and filename, and it uses `codebase`. In the PageData example, the JAR file is in http://cadenhead.org/book/java21days/java/PageData.jar.

The `application-desc` element indicates the application's main class file and any arguments that should be used when that class is executed.

The `main-class` attribute identifies the name of the class file, which is specified without the `.class` file extension.

If the class should be run with one or more arguments, place `argument` elements within an opening `<application-desc>` tag and a closing `</application-desc>` tag.

The following XML specifies that the `PageData` class should be run with two arguments: `http://java.com` and `yes`:

Click here to view code image

```
<application-desc main-class="PageData">
  <argument>http://java.com</argument>
  <argument>yes</argument>
</application-desc>
```

You can test the PageData application from my web server or upload it to your own and edit the JNLP file accordingly.

If you are trying it on your own server, after you have created the `PageData.jnlp` file, change line 5 of Listing 14.2 so that it refers to the folder on a web server where your application's JAR file, icon file, and JNLP file will be stored.

Upload all three of the project's files to this folder, and then run your browser and load the JNLP file using its full web address. If your web server is configured to support Java Web Start, the application is loaded and begins running., Figure 14.5 shows the output if the web address `http://www.java.com/` is requested.

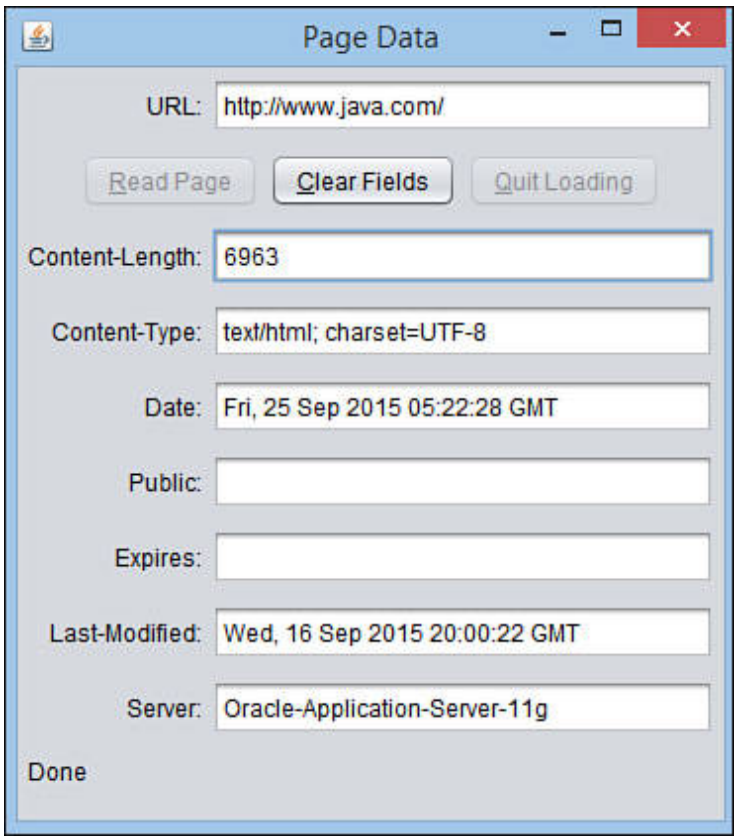


FIGURE 14.5 Running PageData using Java Web Start.

For this application to be run without restriction, the `PageData.jar` file must be digitally signed. For real-world applications, this requires the services of a certificate-granting authority.

For testing purposes, there are two programs included with the JDK that can be used to create a key and use it to digitally sign a JAR file. To find out how to use them, read Appendix E, "Programming with the Java Development Kit."

A self-signed JAR file only is suitable for your own testing purposes. No current web browser will accept that the JAR file is secure or allow users to run it.

Note

The only way to establish that your Java Web Start application is trustworthy is to go through one of the professional certificate-granting companies, prove your identity, and purchase a code-signing certificate. This certificate will be used when signing your JAR files. These companies are called certificate authorities.

California offers a list of digital signature certification authorities at www.tinyurl.com/califsign that's a useful resource for any programmer who needs code-signing certificates, not just in that state.

At the time of this writing, the lowest-priced code-signing certificates were $95 per year from K Software at http://codesigning.ksoftware.net. Most other authorities charge $200 or more.

Supporting Web Start on a Server

If your server does not support Java Web Start, you might see the text of your JNLP file loaded in a page, and the application will not open.

A web server must be configured to recognize that JNLP files are a new type of data that should cause a Java application to run. This is usually accomplished by setting the MIME type associated with files that have the extension JNLP.

MIME, which is an acronym for Multipurpose Internet Mail Extensions, is a protocol for defining Internet content such as email messages, attached files, and any file that can be delivered by a web server.

On an Apache web server, the server administrator can support JNLP by adding the following line to the server's `mime.types` (or `.mime.types`) file:

Click here to view code image

```
application/x-java-jnlp-file JNLP
```

If you can't get Java Web Start working on your server, you can test this project on the book's official site. Load the web page http://cadenhead.org/book/java-21-days/java/PageData.jnlp.

Caution

Java Web Start applications should look exactly like applications do when run by other means. However, there appear to be a few bugs in how much space is allocated to components on a graphical user interface. On a Windows system, you might need to add 50 pixels to the height of an application before employing it in Java Web Start. Otherwise, the text fields are not tall enough to display numbers.

Additional JNLP Elements

The JNLP format has other elements that can affect the performance of Java Web Start.

It can be used to change the title graphic that appears when the application is launched, run signed applications that have different security privileges, run an application using different versions of the JVM, and other options.

Security

By default, all Java Web Start applications are denied access to some features of a user's computer unless the user has given permission. This is similar to how the functionality of applets is limited.

If your application's JAR file has been digitally signed to verify its authenticity, you can run it without these security restrictions by using the `security` element.

This element is placed inside the `jnlp` element, and it contains one element of its own: `all-permissions`. To employ the normal security restrictions for an application run by Web Start, add this to a JNLP file:

Click here to view code image

```
<security>
  <j2ee-application-client-permission/>
</security>
```

If the tag `all-permissions` had been used instead, the application would have no security restrictions. This has major implications for user security, so it should be done with care.

Descriptions

If you want to provide more information about your application for users of Java Web Start, you can place one or more `description` elements inside the `information` element.

Four kinds of descriptions can be provided using the `kind` attribute of the `description` element:

- `kind="one-line"`—A succinct one-line description, used in lists of Web Start applications
- `kind="short"`—A paragraph-long description, used when space is available
- `kind="tooltip"`—A ToolTip description
- No `kind` attribute—A default description, used for any other descriptions not specified

All these are optional. Here's an example that provides descriptions for the `PageData` application:

Click here to view code image

```
<description>The PageData application.</description>
<description kind="one-line">An application to learn more about
web
servers and pages.</description>
<description kind="tooltip">Learn about web servers and
pages.</description>
<description kind="short">PageData, a simple Java application
that
takes a URL and displays information about the URL and the web
server that delivered it.</description>
```

Icons

The PageData JNLP file includes a 64×64 icon, `pagedataicon.gif`, used in two different ways:

- When the PageData application is being loaded by Java Web Start, the icon is displayed in a window next to the program's name and author.
- If a PageData icon is added to a user's desktop, the icon is used at a different size: 32×32.

When an application is loading, you can use a second `icon` element to specify a graphic that will be displayed in place of the icon, title, and author.

This graphic is called the application's *splash screen,* and it is specified with the `kind="splash"` attribute, as in this example:

Click here to view code image

```
<icon kind="splash" href="pagedatasplash.gif" width="300"
height="200" />
```

The `width` and `height` attributes, which also can be used with the other kind of icon graphic, specify the image's display size in pixels.

This second `icon` element should be placed inside the `information` element.

Note

For more information on using the technology with your own applications, visit Oracle's Java Web Start site: http://oracle.com/technetwork/java/javase/javawebstart

---

Improving Performance with SwingWorker

The responsiveness of a Swing application depends largely on how well the software handles time-consuming tasks in response to user input.

Applications ordinarily execute tasks in one thread. So if something takes a long time to accomplish, such as loading a large file or parsing data from an XML document, the user might notice a lag in performance while this is taking place.

Swing programs also require all user-interface components to be running within the same thread.

The best way to take care of both requirements is to use `SwingWorker`, a class in the `javax.swing` package that's designed to run time-consuming tasks in their own worker thread and report the results.

`SwingWorker` is an abstract class that must be subclassed by applications that require a worker:

Click here to view code image

```
public class DiceWorker extends SwingWorker {
    // body of class
}
```

The `doInBackground()` method should be overridden in the new class to perform the task.

Today's next project is a Swing application that rolls three six-sided dice a user-selected number of times and tabulates the results. Sixteen text fields represent the possible values, which range from 3 to 18.

The application is developed as two classes: the `DiceRoller` frame, which holds the graphical user interface, and the `DiceWorker` Swing worker, which handles the dice rolls.

Because the application allows the user to roll the dice thousands or even millions of times, putting this task in a worker keeps the Swing interface responsive to user input.

Listing 14.3 contains the worker class, `DiceWorker`. Create this as an empty Java file in NetBeans with that class name and the package `com.java21days`. In the New File dialog, be sure the Project selected is Java21, not PageData.

LISTING 14.3 The Full Text of `DiceWorker.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class DiceWorker extends SwingWorker {
 6:     int timesToRoll;
 7:
 8:     // set up the Swing worker
 9:     public DiceWorker(int timesToRoll) {
10:         super();
11:         this.timesToRoll = timesToRoll;
12:     }
13:
14:     // define the task the worker performs
15:     protected int[] doInBackground() {
16:         int[] result = new int[16];
17:         for (int i = 0; i < this.timesToRoll; i++) {
18:             int sum = 0;
19:             for (int j = 0; j < 3; j++) {
20:                 sum += Math.floor(Math.random() * 6);
21:             }
22:             result[sum] = result[sum] + 1;
23:         }
```

```
24:          // transmit the result
25:          return result;
26:      }
27: }
```

There's no way to do anything with this class until you create the next one, `DiceRoller`.

A Swing worker needs only one method, `doInBackground()`, which performs the task in the background. The method must use the `protected` level of access control and return a value produced by the work. `DiceWorker` creates a 16-element integer array that contains dice-roll results.

Another class can use this worker in three steps:

1. Call the worker's `DiceWorker(`*`int`*`)` constructor with the number of rolls as the argument.
2. Call the worker's `addPropertyChangeListener(`*`Object`*`)` method to add a listener that will be notified when the task is complete.
3. Call the worker's `execute()` method to begin the work.

The `execute()` method causes the worker's `doInBackground()` method to be called.

A property change listener is an event listener from `java.beans`, the JavaBeans package that establishes ways in which components on a user interface can interact with each other.

In this case, a Swing worker wants to announce that its work is finished, which could take place long after the worker began its work. Listeners are the best way to handle notifications of this kind because they free a graphical user interface to handle other things.

The property change listener interface has one method:

Click here to view code image

```
public void propertyChange(PropertyChangeEvent event) {
    // ...
}
```

The `DiceRoller` class in the `com.java21days` package, shown in Listing 14.4, presents a graphical user interface that can display dice-roll results and begin a set of rolls.

LISTING 14.4 The Full Text of `DiceRoller.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import java.beans.*;
 6: import javax.swing.*;
 7:
 8: public class DiceRoller extends JFrame implements
ActionListener,
 9:     PropertyChangeListener {
10:
11:     // the table for dice-roll results
12:     JTextField[] total = new JTextField[16];
13:     // the "Roll" button
14:     JButton roll;
15:     // the number of times to roll
16:     JTextField quantity;
17:     // the Swing worker
18:     DiceWorker worker;
19:
20:     public DiceRoller() {
21:         super("Dice Roller");
22:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23:         setLookAndFeel();
24:         setSize(850, 145);
25:
26:         // set up top row
27:         JPanel topPane = new JPanel();
28:         GridLayout paneGrid = new GridLayout(1, 16);
29:         topPane.setLayout(paneGrid);
30:         for (int i = 0; i < 16; i++) {
31:             // create a textfield and label
32:             total[i] = new JTextField("0", 4);
33:             JLabel label = new JLabel((i + 3) + ": ");
34:             // create this cell in the grid
35:             JPanel cell = new JPanel();
36:             cell.add(label);
37:             cell.add(total[i]);
38:             // add the cell to the top row
39:             topPane.add(cell);
40:         }
41:
42:         // set up bottom row
43:         JPanel bottomPane = new JPanel();
44:         JLabel quantityLabel = new JLabel("Times to Roll:
");
```

```
45:          quantity = new JTextField("0", 5);
46:          roll = new JButton("Roll");
47:          roll.addActionListener(this);
48:          bottomPane.add(quantityLabel);
49:          bottomPane.add(quantity);
50:          bottomPane.add(roll);
51:
52:          // set up frame
53:          GridLayout frameGrid = new GridLayout(2, 1);
54:          setLayout(frameGrid);
55:          add(topPane);
56:          add(bottomPane);
57:
58:          setVisible(true);
59:      }
60:
61:      // respond when the "Roll" button is clicked
62:      public void actionPerformed(ActionEvent event) {
63:          int timesToRoll;
64:          try {
65:              // turn off the button
66:              timesToRoll =
Integer.parseInt(quantity.getText());
67:              roll.setEnabled(false);
68:              // set up the worker that will roll the dice
69:              worker = new DiceWorker(timesToRoll);
70:              // add a listener that monitors the worker
71:              worker.addPropertyChangeListener(this);
72:              // start the worker
73:              worker.execute();
74:          } catch (Exception exc) {
75:              System.out.println(exc.getMessage());
76:              exc.printStackTrace();
77:          }
78:      }
79:
80:      // respond when the worker's task is complete
81:      public void propertyChange(PropertyChangeEvent event) {
82:          try {
83:              // get the worker's dice-roll results
84:              int[] result = (int[]) worker.get();
85:              // store the results in text fields
86:              for (int i = 0; i < result.length; i++) {
87:                  total[i].setText("" + result[i]);
88:              }
89:          } catch (Exception exc) {
90:              System.out.println(exc.getMessage());
91:              exc.printStackTrace();
92:          }
93:      }
94:
```

```
 95:     private static void setLookAndFeel() {
 96:         try {
 97:             UIManager.setLookAndFeel(
 98:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
 99:             );
100:         } catch (Exception exc) {
101:             // ignore error
102:         }
103:     }
104:
105:     public static void main(String[] arguments) {
106:         new DiceRoller();
107:     }
108: }
```

This class can be run as an application. Choose Run, Run File in NetBeans.

Most of `DiceRoller` creates and lays out the user-interface components: 16 text fields, a Times to Roll text field, and a Roll button.

The `actionPerformed()` method responds to a click of the Roll button by creating a Swing worker that will roll the dice, adding a property change listener and starting work.

Calling `worker.execute()` in line 73 causes the worker's `doInBackground()` method to be called.

When the worker is finished rolling the dice, the `propertyChange()` method of `DiceRoller` receives a property change event.

This method receives the result of `doInBackground()` by calling the worker's `get()` method (line 84), which must be cast to an integer array:

Click here to view code image

```
int[] result = (int[] worker.get();
```

The application is shown in Figure 14.6.

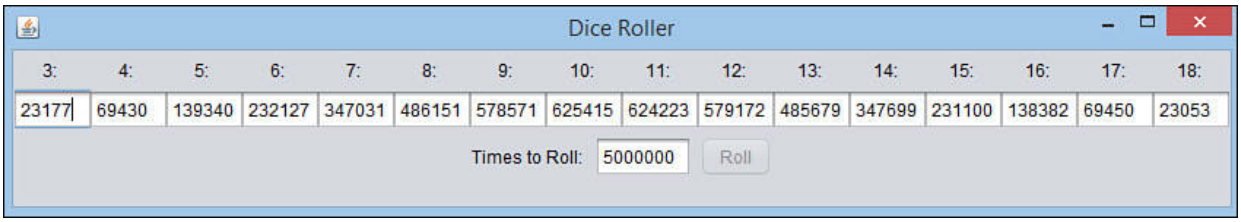


FIGURE 14.6 Tabulating dice-roll results prepared by `DiceWorker`.

Summary

The topics covered today are two capabilities that enhance Java's capabilities for application development: browser-based program deployment and Swing performance improvements through the use of threads.

With Java Web Start, users no longer need to run an installation program to set up a Java application and the JVM that executes the class. Web Start takes care of this automatically, after the user's browser has been equipped to use the Java Runtime Environment.

Support for Web Start is offered through the Java Network Launching Protocol (JNLP), an XML file format used to define and set up Java Web Start.

The `SwingWorker` class improves Swing application performance by putting a time-consuming task in its own thread. The class handles all the work required to start and stop the thread behind the scenes.

When you create a subclass of `SwingWorker`, you can focus on the task that must be performed.

Q&A

Q I have written a Java applet that I want to make available using Java Web Start. Should I convert it to an application or go ahead and run it as is?

A If you would be converting your program to an application simply to run it with Web Start, that's probably unnecessary. The purpose of the `applet-desc` tag is to make it possible to run applets without modification in Java Web Start. The only reason to undertake the conversion is if you want to change other things about your program, such as the switch from `init()` to a constructor method.

Q How can I make sure that a `SwingWorker` object has finished working?

A Call the worker's `isDone()` method, which returns `true` when the task has finished executing.

Note that this method returns `true` no matter how the task completes. So if it is canceled or interrupted or fails in some other manner, it returns `true`.

The `isCancelled()` method can be used to check whether the task was canceled.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What interface must be implemented for you to be notified when a `SwingWorker` has finished executing?

   A. `ActionListener`

   B. `PropertyChangeListener`

   C. `SwingListener`

2. Which XML element is used to identify the name, author, and other details about a Java Web Start–run application?

   A. `jnlp`

   B. `information`

   C. `resources`

3. What security restrictions apply to a Java Web Start application?

   A. There are no restrictions.

   B. The same restrictions that are in place for applications

   C. The restrictions chosen by the user

Answers

1. B. The `PropertyChangeListener` in the `java.beans` package receives a `propertyChange()` event when the worker finishes.
2. B. The application is described using elements contained within an opening `<information>` tag and a closing `</information>` tag.
3. C. A Java Web Start application has few restrictions. They are limited to important functionality such as saving files or opening Internet connections. These restrictions are dropped if a user explicitly grants those privileges as the application runs.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
import java.awt.*;
import javax.swing.*;

public class SliderFrame extends JFrame {
    public SliderFrame() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JSlider value = new JSlider(0, 255, 100);
        setSize(325, 150);
        setVisible(true);
    }

    public static void main(String[] arguments) {
        new SliderFrame();
    }
}
```

What will happen when you attempt to compile and run this source code?

A. It compiles without error and runs correctly.

B. It compiles without error but does not display anything in the frame.

C. It does not compile because the content pane is empty.

D. It does not compile because of the `new SliderFrame()` statement.

The answer is available on the book's website at www.java21days.com. Visit the Day 14 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Turn one of the applications created during the first two weeks into one that can be launched with Java Web Start.
2. Create a new JNLP file that runs the PageData application using version 1.3 of the JVM, and force users to be connected to the Internet when it is run.

Exercise solutions are offered on the book's website at www.java21days.com.

Week III: Java Programming

15 Working with Input and Output

16 Using Inner Classes and Closures

17 Communicating Across the Internet

18 Accessing Databases with JDBC 4.2 and Derby

19 Reading and Writing RSS Feeds

20 XML Web Services

21 Writing Android Apps with Java

Day 15. Working with Input and Output

Many of the programs you create with Java need to interact with some kind of data source. Information can be stored on a computer in many ways, including files on a hard drive or DVD, pages on a website, and even bytes in the computer's memory.

You might expect to need a different technique to handle each different storage device. Fortunately, that isn't the case.

In Java, information can be stored and retrieved using a communications system called streams, which are implemented in the `java.io` package and are enhanced by the `java.nio.file` package.

Today, you learn how to create input streams to read information and output streams to store information. You work with the following:

- Byte streams, which are used to handle bytes, integers, and other simple data types
- Character streams, which handle text files and other text sources

You can deal with all data in the same way when you know how to work with an input stream, whether the information is coming from a disk, the Internet, or even another program. The same is true of using output streams to transmit data.

Introduction to Streams

In Java, all data is written and read using streams. Streams, like the bodies of water that share the same name, carry something from one place to another.

A stream is a path traveled by data in a program. An input stream sends data from a source into a program, and an output stream sends data from a program to a destination.

You will deal with two types of streams today: byte streams and character streams. *Byte streams* carry integers with values that range from 0 to 255. A diverse assortment of data can be expressed in byte format, including numeric data, executable programs, Internet communications, and bytecode—the class files run by a Java Virtual Machine (JVM).

In fact, every kind of data imaginable can be expressed using either individual bytes or a series of bytes combined.

Character streams are a specialized type of byte stream that handles only textual data. They're distinguished from byte streams because Java's character set supports Unicode, a standard that includes many more characters than could be expressed easily using bytes.

Any kind of data that involves text should use character streams, including text files, web pages, and other common types of text.

Using a Stream

The procedure for using either a byte stream or character stream in Java is largely the same. Before you start working with the specifics of the `java.io` and `java.nio.file` classes, it's useful to walk through the process of creating and using streams.

For an input stream, the first step is to create an object associated with the data source. For example, if the source is a file on your hard drive, a `FileInputStream` object could be associated with this file.

After you have a stream object, you can read information from that stream by using one of the object's methods. `FileInputStream` includes a `read()` method that returns a byte read from the file.

When you're finished reading information from the stream, you call the `close()` method to indicate that you're finished using the stream.

For an output stream, you begin by creating an object associated with the data's destination. One such object can be created from the `BufferedWriter` class, which represents an efficient way to create text files.

The `write()` method is the simplest way to send information to the output stream's destination. For instance, a `BufferedWriter write()` method can send individual characters to an output stream.

As with input streams, the `close()` method is called on an output stream when you have no more information to send.

Filtering a Stream

The simplest way to use a stream is to create it and then call its methods to send or receive data, depending on whether it's an output stream or input stream.

Many of the classes you will work with today achieve more sophisticated results when a filter is associated with a stream before reading or writing any data.

A *filter* is a type of stream that modifies how an existing stream is handled. Think of a dam on a mountain stream. The dam regulates the flow of water from the points upstream to the points downstream. The dam is a type of filter. Remove it, and the water would flow in a less-controlled fashion.

The procedure for using a filter on a stream is as follows:

1. Create a stream associated with a data source or data destination.
2. Associate a filter with that stream.
3. Read data from or write data to the filter rather than the original stream.

The methods you call on a filter are the same as the methods you would call on a stream. There are `read()` and `write()` methods, just as there would be on an unfiltered stream.

You even can associate a filter with another filter, so the following path for information is possible: An input stream associated with a text file is filtered through a Spanish-to-English translation filter, which is then filtered through a no-profanity filter. Finally, it is sent to its destination—a human being who wants to read it.

If this is confusing in the abstract, you will have opportunities to see the process in practice in the following sections.

Handling Exceptions

Several exceptions in the `java.io` package might occur when you are working with files and streams. Two common ones are `FileNotFoundException` and `EOFException`.

A `FileNotFoundException` occurs when you try to create a stream or file object using a file that couldn't be located.

An `EOFException` indicates that the end of a file has been reached unexpectedly as data was being read from the file through an input stream.

These exceptions are subclasses of `IOException`. One way to deal with all of them is to enclose all input and output statements in a `try-catch` block that catches `IOException` objects. Call the exception's `toString()` or `getMessage()` methods in the `catch` block to find out more about the problem.

Byte Streams

All byte streams are a subclass of either `InputStream` or `OutputStream`. These classes are abstract, so you cannot create a stream by creating objects of these classes directly. Instead, you create streams through one of their subclasses, such as the following:

- `FileInputStream` and `FileOutputStream` are byte streams stored in files on disk, CD, or other storage devices.
- `DataInputStream` and `DataOutputStream` are a filtered byte stream from which data such as integers and floating-point numbers can be read.

`InputStream` is the superclass of all input streams.

File Streams

The byte streams you'll work with most often are likely to be file streams. They are used to exchange data with files on your disk drives, CDs, or other storage devices you can refer to by using a folder path and filename.

You can send bytes to a file output stream and receive bytes from a file input stream.

File Input Streams

A file input stream can be created with the `FileInputStream(`*`String`*`)` constructor. The *`String`* argument should be the filename. You can include a path reference with the filename, which enables the file to be in a different folder from the class loading it. The following statement creates a file input stream from the file `scores.dat`:

Click here to view code image

```
FileInputStream fis = new FileInputStream("scores.dat");
```

Path references can be indicated in a manner specific to a platform, such as this example to read a file on a Windows system:

Click here to view code image

```
FileInputStream f1 = new
FileInputStream("C:\\data\\calendar.txt");
```

Note

Because Java uses backslash characters in escape codes, the code `\\` must be used in place of `\` in path references in Windows.

Here's a Linux example:

Click here to view code image

```
FileInputStream f2 = new FileInputStream("/data/calendar.txt");
```

A better way to refer to paths is to use the class variable `separator` in the `File` class, which works on any operating system:

Click here to view code image

```
char sep = File.separator;
FileInputStream f2 = new FileInputStream(sep + "data"
    + sep + "calendar.txt");
```

After you create a file input stream, you can read bytes from the stream by calling its `read()` method. This method returns an integer containing the next byte in the stream. The method returns –1, which is not a possible byte value, when the end of the file stream has been reached.

To read more than one byte of data from the stream, call its `read(byte[], int, int)` method. The arguments to this method are as follows:

- A byte array where the data will be stored
- The element inside the array where the data's first byte should be stored
- The number of bytes to read

Unlike the other `read()` method, this does not return data from the stream. Instead, it returns either an integer that represents the number of bytes read or –1 if no bytes were read before the end of the stream was reached.

The following statements use a `while` loop to read the data in a `FileInputStream` object called `diskfile`:

Click here to view code image

```
int newByte = 0;
while (newByte != -1) {
    newByte = diskfile.read();
    System.out.print(newByte + " ");
}
```

This loop reads the entire file referenced by `diskfile` one byte at a time and displays each byte, followed by a space character. It also displays –1 when the end of the file is reached; you could guard against this easily with an `if` statement.

The ByteReader application, shown in in Listing 15.1, uses a similar technique to read a file input stream. The input stream's `close()` method is used to close the stream after the last byte in the file is read. Always close streams when you no longer need them; doing so frees system resources. Create the `ByteReader` class in the `com.java21days` package as an empty Java file in NetBeans.

LISTING 15.1 The Full Text of `ByteReader.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class ByteReader {
 6:     public static void main(String[] arguments) {
 7:         try (
 8:             FileInputStream file = new
 9:                 FileInputStream("save.gif")
10:             ) {
11:
12:             boolean eof = false;
13:             int count = 0;
14:             while (!eof) {
15:                 int input = file.read();
16:                 System.out.print(input + " ");
17:                 if (input == -1)
18:                     eof = true;
19:                 else
20:                     count++;
21:             }
22:             file.close();
23:             System.out.println("\nBytes read: " + count);
24:         } catch (IOException e) {
25:             System.out.println("Error -- " + e.toString());
26:         }
27:     }
28: }
```

This application reads the byte data from the `save.gif` file in the main folder of the Java21 project. That file was used during Day 10, "Building a Swing Interface."

When you run the program, each byte in `save.gif` is displayed, followed by a count of the total number of bytes. Figure 15.1 shows the output.

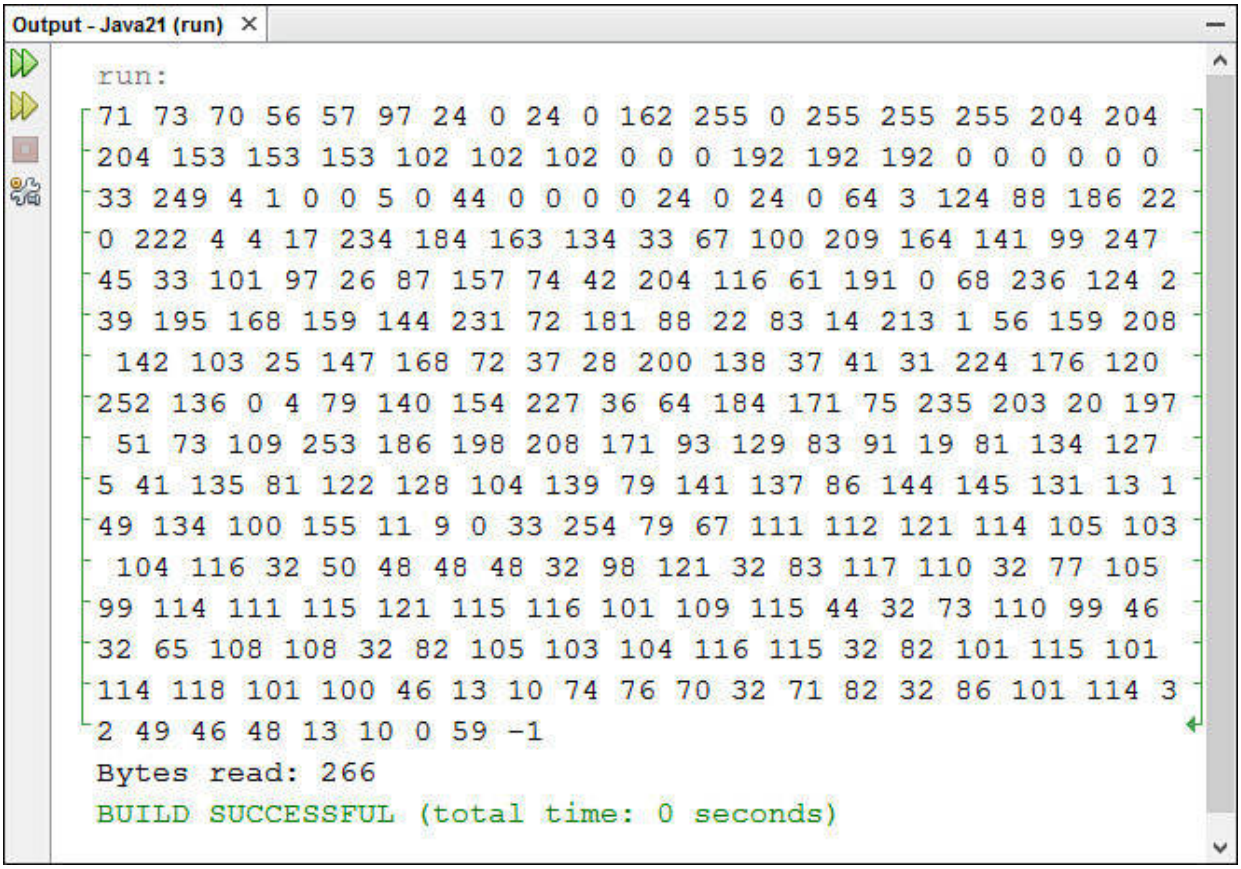


FIGURE 15.1 Reading byte data from a file.

File Output Streams

A file output stream can be created with the `FileOutputStream(String)` constructor. The usage is the same as with the `FileInputStream(String)` constructor, so you can specify a path along with a filename.

You have to be careful when specifying the file associated with an output stream. If it's the same as an existing file, the original is wiped out when you start writing data to the stream.

You can create a file output stream that appends data after the end of an existing file with the `FileOutputStream (String, boolean)`

constructor. The string specifies the file, and the Boolean argument should equal `true` to append data instead of overwriting existing data.

The file output stream's `write(int)` method is used to write bytes to the stream. After the last byte has been written to the file, the stream's `close()` method closes the stream.

To write more than one byte, you can use the `write(byte[], int, int)` method. This works in a manner similar to the `read(byte[], int, int)` method described previously. The arguments to this method are the byte array containing the bytes to output, the starting point in the array, and the number of bytes to write.

The ByteWriter application, shown in Listing 15.2, writes an integer array to a file output stream. Create it in NetBeans in the `com.java21days` package.

LISTING 15.2 The Full Text of `ByteWriter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class ByteWriter {
 6:     public static void main(String[] arguments) {
 7:         int[] data = { 71, 73, 70, 56, 57, 97, 13, 0, 12, 0,
145,
 8:             0, 0, 255, 255, 255, 255, 255, 0, 0, 0, 0, 0, 0,
0, 44,
 9:             0, 0, 0, 0, 13, 0, 12, 0, 0, 2, 38, 132, 45, 121,
11,
10:             25, 175, 150, 120, 20, 162, 132, 51, 110, 106,
239, 22,
11:             8, 160, 56, 137, 96, 72, 77, 33, 130, 86, 37,
219, 182,
12:             230, 137, 89, 82, 181, 50, 220, 103, 20, 0, 59 };
13:         try (FileOutputStream file = new
14:             FileOutputStream("pic.gif")) {
15:
16:             for (int i = 0; i < data.length; i++) {
17:                 file.write(data[i]);
18:             }
19:             file.close();
20:         } catch (IOException e) {
21:             System.out.println("Error -- " + e.toString());
```

```
22:             }
23:         }
24: }
```

The following things take place in this program:

- Lines 7–12 create an integer array called `data` and fill it with elements.
- Lines 13–14 create a file output stream with the filename `pic.gif` in the main project folder in NetBeans.
- Lines 16–18 use a `for` loop to cycle through the `data` array and write each element to the file stream.
- Line 19 closes the file output stream.

The `FileOutputStream` object is created inside the parentheses of the `try` statement to make sure its resources are freed up when the block finishes executing, even in case of an error.

After you run this program, you can display the `pic.gif` file in any web browser or graphics-editing tool. It's a small image file in GIF format, as shown in Figure 15.2.

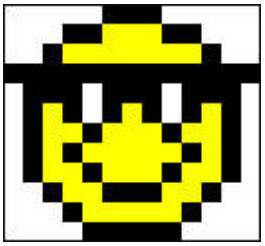


FIGURE 15.2 The `pic.gif` file (enlarged).

Filtering a Stream

Filtered streams are streams that modify the information sent through an existing stream. They are created using the subclasses `FilterInputStream` and `FilterOutputStream`.

These classes do not handle any filtering operations themselves. Instead, they have subclasses, such as `BufferInputStream` and `DataOutputStream`, which handle specific types of filtering.

Byte Filters

Information is delivered more quickly if it can be sent in large chunks, even if those chunks are received faster than they can be handled.

For example, consider which of the following book-reading techniques is faster:

- A friend lends you a book, and you read it.
- A friend lends you a book one page at a time and doesn't give you a new page until you have finished the previous one.

Obviously, the first technique is faster and more efficient. The same benefits are true of buffered streams in Java.

A *buffer* is a storage place where data can be kept before it is needed by a program that reads or writes that data. By using a buffer, you can get data without always going back to the original source of the data.

Buffers are essential when reading extremely large files. Without them, the data from the file could take up all of a Java virtual machine's memory.

Buffered Streams

A buffered input stream fills a buffer with data that hasn't been handled yet. When a program needs this data, it looks to the buffer before going to the original stream source.

Buffered byte streams use the `BufferedInputStream` and `BufferedOutputStream` classes.

A buffered input stream is created using one of the following constructors:

- `BufferedInputStream(`*`InputStream`*`)` creates a buffered input stream for the specified *`InputStream`* object.
- `BufferedInputStream(`*`InputStream`*`, int)` creates the specified *`InputStream`* buffered stream with a buffer of size *`int`*.

The simplest way to read data from a buffered input stream is to call its `read()` method with no arguments. This action normally returns an integer from 0 to 255 representing the next byte in the stream. If the end of the stream has been reached and no byte is available, –1 is returned.

You also can use the `read(`*`byte`*`[], int, int)` method available for other input streams, which loads stream data into a byte array.

A buffered output stream is created using one of these two constructors:

- `BufferedOutputStream(OutputStream)` creates a buffered output stream for the specified *`OutputStream`* object.
- `BufferedOutputStream(OutputStream, int)` creates the specified *`OutputStream`* buffered stream with a buffer of size *`int`*.

The output stream's `write(int)` method can be used to send a single byte to the stream, and the `write(byte[], int, int)` method writes multiple bytes from the specified byte array. The arguments to this method are the byte array, array starting point, and number of bytes to write.

Note

Although the `write()` method takes an integer as input, the value should be from 0 to 255. If you specify a number higher than 255, it is stored as the remainder of the number divided by 256. You can test this when running the project you will create later today.

When data is directed to a buffered stream, it is not output to its destination until the stream fills or the buffered stream's `flush()` method is called.

The next project, the BufferDemo application, writes a series of bytes to a buffered output stream associated with a text file. The first and last integers in the series are specified as two arguments.

After writing to the text file, BufferDemo creates a buffered input stream from the file and reads the bytes back in. Listing 15.3 contains the source code. Put this class in the `com.java21days` package when creating it in NetBeans.

LISTING 15.3 The Full Text of `BufferDemo.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class BufferDemo {
 6:     public static void main(String[] arguments) {
 7:         int start = 0;
 8:         int finish = 255;
 9:         if (arguments.length > 1) {
10:             start = Integer.parseInt(arguments[0]);
11:             finish = Integer.parseInt(arguments[1]);
```

```
12:         } else if (arguments.length > 0) {
13:             start = Integer.parseInt(arguments[0]);
14:         }
15:         ArgStream as = new ArgStream(start, finish);
16:         System.out.println("\nWriting: ");
17:         boolean success = as.writeStream();
18:         System.out.println("\nReading: ");
19:         boolean readSuccess = as.readStream();
20:     }
21: }
22:
23: class ArgStream {
24:     int start = 0;
25:     int finish = 255;
26:
27:     ArgStream(int st, int fin) {
28:         start = st;
29:         finish = fin;
30:     }
31:
32:     boolean writeStream() {
33:         try (FileOutputStream file = new
34:                 FileOutputStream("numbers.dat");
35:             BufferedOutputStream buff = new
36:                 BufferedOutputStream(file)) {
37:
38:             for (int out = start; out <= finish; out++) {
39:                 buff.write(out);
40:                 System.out.print(" " + out);
41:             }
42:             buff.close();
43:             return true;
44:         } catch (IOException e) {
45:             System.out.println("Exception: " +
e.getMessage());
46:             return false;
47:         }
48:     }
49:
50:     boolean readStream() {
51:         try (FileInputStream file = new
52:                 FileInputStream("numbers.dat");
53:             BufferedInputStream buff = new
54:                 BufferedInputStream(file)) {
55:
56:             int in;
57:             do {
58:                 in = buff.read();
59:                 if (in != -1) {
60:                     System.out.print(" " + in);
61:                 }
```

```
62:             } while (in != -1);
63:             System.out.println();
63:             buff.close();
64:             return true;
65:         } catch (IOException e) {
66:             System.out.println("Exception: " +
e.getMessage());
67:             return false;
68:         }
69:     }
70: }
```

This program's output depends on the two arguments specified when it was run. If you use 3 and 19, the output in Figure 15.3 is shown.

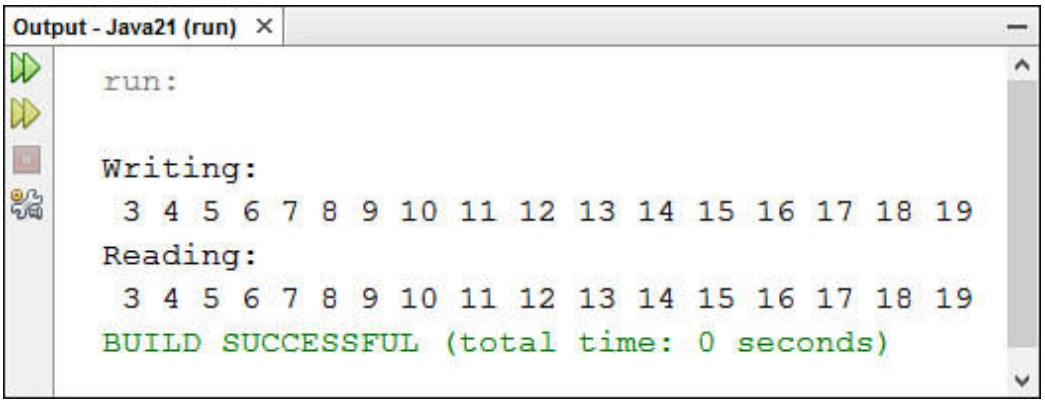


FIGURE 15.3 Reading and writing buffered streams.

It also can be run without arguments, using 1 and 255 as default values.

This application consists of two classes: `BufferDemo` and a helper class called `ArgStream`. `BufferDemo` gets the two arguments' values, if they are provided, and uses them in the `ArgStream()` constructor.

The `writeStream()` method of `ArgStream` is called in line 17 to write the series of bytes to a buffered output stream, and the `readStream()` method is called in line 19 to read back those bytes.

Even though they are moving data in two directions, the `writeStream()` and `readStream()` methods are substantially the same. They take the following format:

- The filename, `numbers.dat`, is used to create a file input or output stream.
- The file stream is used to create a buffered input or output stream.

- The buffered stream's `write()` method is used to send data, or the `read()` method is used to receive data.
- The buffered stream is closed.

Because file streams and buffered streams throw `IOException` objects if an error occurs, all operations involving the streams are enclosed in a `try-catch` block for this exception.

Note

The `Boolean` return values in `writeStream()` and `readStream()` indicate whether the stream operation was completed successfully. They aren't used in this program, but it's good practice to let callers of these methods know if something goes wrong. When the value is `false`, the operation could be attempted again.

Console Input Streams

One of the things many experienced programmers miss when they begin learning Java is the ability to read textual or numeric input from the console while running an application. No input method is comparable to the output methods `System.out.print()` and `System.out.println()`.

Now that you can work with buffered input streams, you can put them to use receiving console input.

The `System` class, part of the `java.lang` package, has a class variable called `in` that is an `InputStream` object. This object receives input from the keyboard through the stream.

You can work with this stream as you would any other input stream. The following statement creates a new buffered input stream associated with the `System.in` input stream:

Click here to view code image

```
BufferedInputStream command = new BufferedInputStream(System.in);
```

The next project, the `ConsoleInput` class, contains a class method you can use to receive console input in any of your Java applications. Enter the code shown in Listing 15.4 in NetBeans, making sure to put it in the package `com.java21days`.

LISTING 15.4 The Full Text of `ConsoleInput.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class ConsoleInput {
 6:     public static String readLine() {
 7:         StringBuilder response = new StringBuilder();
 8:         try (BufferedInputStream buff = new
 9:             BufferedInputStream(System.in)) {
10:
11:             int in;
12:             char inChar;
13:             do {
14:                 in = buff.read();
15:                 inChar = (char) in;
16:                 if ((in != -1) & (in != '\n') & (in != '\r'))
{
17:                     response.append(inChar);
18:                 }
19:             } while ((in != -1) & (inChar != '\n') & (in !=
'\r'));
20:             buff.close();
21:             return response.toString();
22:         } catch (IOException e) {
23:             System.out.println("Exception: " +
e.getMessage());
24:             return null;
25:         }
26:     }
27:
28:     public static void main(String[] arguments) {
29:         System.out.print("\nWhat is your name? ");
30:         String input = ConsoleInput.readLine();
31:         System.out.println("\nHello, " + input);
32:     }
33: }
```

The `ConsoleInput` class includes a `main()` method that demonstrates how it can be used. When you compile and run it as an application, the output should resemble Figure 15.4.

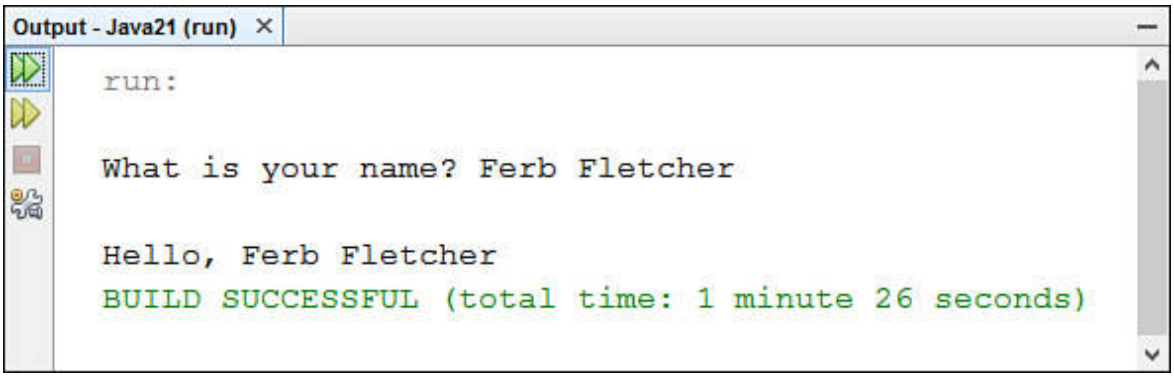


FIGURE 15.4 Reading keyboard input from the console window.

`ConsoleInput` reads user input through a buffered input stream using the stream's `read()` method, which returns –1 when the end of input has been reached. This occurs when the user presses the Enter key, a carriage return (character '\r'), or a newline (character '\n').

Data Streams

If you need to work with data that isn't represented as bytes or characters, you can use data input and data output streams. These streams filter an existing byte stream so that each of the following primitive types can be directly read from or written to the stream: `boolean`, `byte`, `double`, `float`, `int`, `long`, and `short`.

A data input stream is created with the `DataInputStream(`*`InputStream`*`)` constructor. The argument should be an existing input stream such as a buffered input stream or a file input stream.

A data output stream requires the `DataOutputStream(`*`OutputStream`*`)` constructor, which indicates the associated output stream.

The following read and write methods apply to data input and output streams, respectively:

- `readBoolean()`, `writeBoolean(`*`boolean`*`)`
- `readByte()`, `writeByte(`*`integer`*`)`
- `readDouble()`, `writeDouble(`*`double`*`)`
- `readFloat()`, `writeFloat(`*`float`*`)`
- `readInt()`, `writeInt(`*`int`*`)`
- `readLong()`, `writeLong(`*`long`*`)`

- `readShort()`, `writeShort(int)`

Each input method returns the primitive data type indicated by the method's name. For example, the `readFloat()` method returns a `float` value.

There also are `readUnsignedByte()` and `readUnsignedShort()` methods that read in unsigned `byte` and `short` values. Java doesn't support these data types, so they are returned as `int` values.

Note

Unsigned bytes have values ranging from 0 to 255. This differs from Java's `byte` variable type, which ranges from –128 to 127. Along the same lines, an unsigned `short` value ranges from 0 to 65,535, instead of the –32,768 to 32,767 range supported by Java's `short` type.

A data input stream's different read methods do not all return a value that can be used to indicate that the end of the stream has been reached.

As an alternative, you can wait for an `EOFException` (end-of-file exception) to be thrown when a read method reaches the end of a stream. The loop that reads the data can be enclosed in a `try` block, and the associated `catch` statement should handle only `EOFException` objects. You can call `close()` on the stream and take care of other cleanup tasks inside the `catch` block.

This is demonstrated in the next project. Listings 15.5 and 15.6 contain two programs that use data streams. The PrimeWriter application writes the first 400 prime numbers as integers to a file called `400primes.dat`. The PrimeReader application reads the integers from this file and displays them. Both classes are in the `com.java21days` package.

LISTING 15.5 The Full Text of `PrimeWriter.java`

Click here to view code image

```
1: package com.java21days;
2:
3: import java.io.*;
4:
5: public class PrimeWriter {
6:     public static void main(String[] arguments) {
7:         int[] primes = new int[400];
```

```
 8:          int numPrimes = 0;
 9:          // candidate: the number that might be prime
10:          int candidate = 2;
11:          while (numPrimes < 400) {
12:              if (isPrime(candidate)) {
13:                  primes[numPrimes] = candidate;
14:                  numPrimes++;
15:              }
16:              candidate++;
17:          }
18:
19:          try (
20:              // Write output to disk
21:              FileOutputStream file = new
22:                  FileOutputStream("400primes.dat");
23:              BufferedOutputStream buff = new
24:                  BufferedOutputStream(file);
25:              DataOutputStream data = new
26:                  DataOutputStream(buff);
27:              ) {
28:
29:              for (int i = 0; i < 400; i++)
30:                 data.writeInt(primes[i]);
31:              data.close();
32:          } catch (IOException e) {
33:              System.out.println("Error -- " + e.toString());
34:          }
35:      }
36:
37:      public static boolean isPrime(int checkNumber) {
38:          double root = Math.sqrt(checkNumber);
39:          for (int i = 2; i <= root; i++) {
40:              if (checkNumber % i == 0)
41:                  return false;
42:          }
43:          return true;
44:      }
45: }
```

LISTING 15.6 The Full Text of `PrimeReader.java`

Click here to view code image

```
1: package com.java21days;
2:
3: import java.io.*;
4:
5: public class PrimeReader {
```

```
 6:      public static void main(String[] arguments) {
 7:          try (FileInputStream file = new
 8:                  FileInputStream("400primes.dat");
 9:              BufferedInputStream buff = new
10:                  BufferedInputStream(file);
11:              DataInputStream data = new
12:                  DataInputStream(buff)) {
13:
14:              try {
15:                  while (true) {
16:                      int in = data.readInt();
17:                      System.out.print(in + " ");
18:                  }
19:              } catch (EOFException eof) {
20:                  buff.close();
21:              }
22:          } catch (IOException e) {
23:              System.out.println("Error -- " + e.toString());
24:          }
25:      }
26: }
```

Most of the PrimeWriter application is taken up with logic to find the first 400 prime numbers. After you have an integer array containing the first 400 primes, it is written to a data output stream in Listing 15.5 in lines 19–34.

This application is an example of using more than one filter on a stream. The stream is developed in a three-step process:

1. A file output stream associated with a file called `400primes.dat` is created.
2. A new buffered output stream is associated with the file stream.
3. A new data output stream is associated with the buffered stream.

The `writeInt()` method of the data stream is used to write the primes to the file.

The PrimeReader application is simpler because it doesn't need to do anything regarding prime numbers. It just reads integers from a file using a data input stream.

Lines 7–12 of PrimeReader are nearly identical to statements in the PrimeWriter application, except that input classes are used instead of output classes.

The `try-catch` block that handles `EOFException` objects is in lines 14–21 of Listing 15.6. The work of loading the data takes place inside the `try-`

with-resources block, which was introduced during Day 7, "Exceptions and Threads." This approach ensures the input stream objects will be closed properly when no longer needed.

The `while(true)` statement creates an endless loop. This isn't a problem; an `EOFException` automatically occurs when the end of the stream is encountered at some point as the data stream is being read. The `readInt()` method in line 16 of Listing 15.6 reads integers from the stream.

The last several lines of the PrimeReader application's output are shown in Figure 15.5.

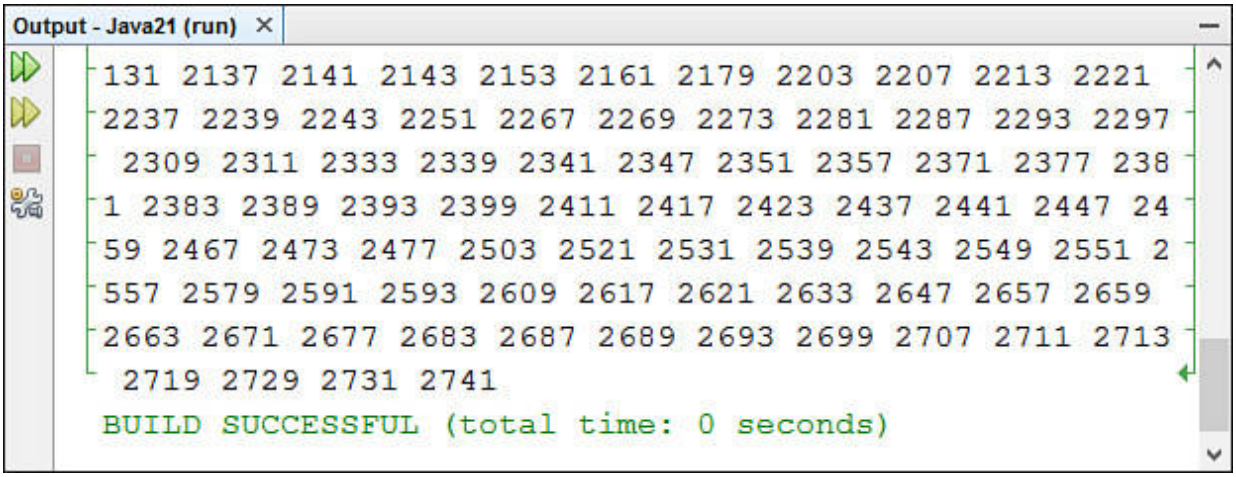


FIGURE 15.5 Reading prime numbers written to a file as integers.

Character Streams

After you know how to handle byte streams, you have most of the skills needed to handle character streams as well. Character streams are used to work with any text represented by the ASCII character set or Unicode, an international character set that includes ASCII.

Examples of files that you can work with through a character stream are plain text files, Web pages, and Java source files.

The classes used to read and write these streams are all subclasses of `Reader` and `Writer`. These should be used for all text input instead of dealing directly with byte streams.

Reading Text Files

`FileReader` is the main class used when reading character streams from a file. This class inherits from `InputStreamReader`, which reads a byte stream and converts the bytes into integer values that represent Unicode characters.

A character input stream is associated with a file using the `FileReader(String)` constructor. The string indicates the file, and it can contain path folder references in addition to a filename.

The following statement creates a new `FileReader` called `look` and associates it with a text file called `index.txt`:

Click here to view code image

```
FileReader look = new FileReader("index.txt");
```

After you have a file reader, you can call the following methods on it to read characters from the file:

- `read()` returns the next character on the stream as an integer.
- `read(char[], int, int)` reads characters into the specified character array with the indicated starting point and number of characters read.

The second method works like similar methods for the byte input stream classes. Instead of returning the next character, it returns either the number of characters that were read or –1 if no characters were read before the end of the stream was reached.

The following method loads a text file using the `FileReader` object `text` and displays its characters:

Click here to view code image

```
FileReader text = new FileReader("readme.txt");
int inByte;
do {
    inByte = text.read();
    if (inByte != -1) {
        System.out.print( (char) inByte );
    }
} while (inByte != -1);
System.out.println("");
text.close();
```

Because a character stream's `read()` method returns an integer, you must cast this to a character before displaying it, storing it in an array, or using it to form a string. Every character has a numeric code that represents its position in the Unicode character set. The integer read from the stream is this numeric code.

If you want to read an entire line of text at a time instead of reading a file character by character, you can use the `BufferedReader` class in conjunction with a `FileReader`.

The `BufferedReader` class reads a character input stream and buffers it for better efficiency. You must have an existing `Reader` object of some kind to create a buffered version. The following constructors can be used to create a `BufferedReader`:

- `BufferedReader(`*`Reader`*`)` creates a buffered character stream associated with the specified `Reader` object, such as `FileReader`.
- `BufferedReader(`*`Reader`*`, `*`int`*`)` creates a buffered character stream associated with the specified `Reader` and with a buffer of size *`int`*.

A buffered character stream can be read using the `read()` and `read(`*`char`*`[], `*`int`*`, `*`int`*`)` methods described for `FileReader`. You can read a line of text using the `readLine()` method.

The `readLine()` method returns a `String` object containing the next line of text on the stream, not including the character or characters that represent the end of a line. If the end of the stream is reached, the value of the string returned equals `null`.

An end-of-line is indicated by any of the following:

- A newline character ('\n')
- A carriage return character ('\r')
- A carriage return followed by a newline ("\n\r")

The project contained in Listing 15.7 is a Java application, SourceReader, that reads its own source file through a buffered character stream. Create it in the `com.java21days` package.

LISTING 15.7 The Full Text of `SourceReader.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class SourceReader {
 6:     public static void main(String[] arguments) {
 7:         try (
 8:             FileReader file = new
 9:                 FileReader("SourceReader.java");
10:             BufferedReader buff = new
11:                 BufferedReader(file)) {
12:
13:             boolean eof = false;
14:             while (!eof) {
15:                 String line = buff.readLine();
16:                 if (line == null) {
17:                     eof = true;
18:                 } else {
19:                     System.out.println(line);
20:                 }
21:             }
22:             buff.close();
23:         } catch (IOException e) {
24:             System.out.println("Error -- " + e.toString());
25:         }
26:     }
27: }
```

Much of this program is comparable to projects created earlier today:

- Lines 8–9: An input source is created: the `FileReader` object associated with the file `SourceReader.java`.
- Lines 10–11: A buffering filter is associated with that input source: the `BufferedReader` object `buff`.
- Lines 13–21: A `readLine()` method is used inside a `while` loop to read the text file one line at a time. The loop ends when the method returns the value `null`.

Before you run the program, make a copy of `SourceReader.java` in the `Java21` project's root folder. To do this, follow these steps:

1. In the Projects pane, right-click `SourceReader.java` and choose Copy. The file is copied to the clipboard.
2. Click the Files pane to bring it to the front.
3. Right-click Java21 at the top of the Files pane; then choose Paste.

A copy will appear in that folder. Run the program to see the SourceReader application's output—the text file `SourceReader.java`.

Writing Text Files

The `FileWriter` class is used to write a character stream to a file. It's a subclass of `OutputStreamWriter`, which has behavior to convert Unicode character codes to bytes.

There are two `FileWriter` constructors: `FileWriter(String)` and `FileWriter(String, boolean)`. The string indicates the name of the file that the character stream will be directed into, which can include a folder path. The optional Boolean argument should equal `true` if the file is to be appended to an existing text file. As with other stream-writing classes, you must be careful not to accidentally overwrite an existing file when you're appending data.

Three methods of `FileWriter` can be used to write data to a stream:

- `write(int)` writes a character.
- `write(char[], int, int)` writes characters from the specified character array with the indicated starting point and number of characters written.
- `write(String, int, int)` writes characters from the specified string with the indicated starting point and number of characters written.

The following example writes a character stream to a file using the `FileWriter` class and the `write(int)` method:

Click here to view code image

```
FileWriter letters = new FileWriter("alphabet.txt");
for (int i = 65; i < 91; i++)
    letters.write( (char) i );
letters.close();
```

The `close()` method is used to close the stream after all characters have been sent to the destination file. The following is the `alphabet.txt` file produced by this code:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The `BufferedWriter` class can be used to write a buffered character stream. This class's objects are created with the `BufferedWriter(Writer)` or `BufferedWriter(Writer, int)`

constructors. The *Writer* argument can be any of the character output stream classes, such as FileWriter. The optional second argument is an integer indicating the size of the buffer to use.

BufferedWriter has the same three output methods as FileWriter: write(*int*), write(*char*[], *int*, *int*), and write(*String*, *int*, *int*).

Another useful output method is newLine(), which sends the preferred end-of-line character (or characters) for the platform being used to run the program.

Tip

The different end-of-line markers can create conversion hassles when files are transferred from one operating system to another, such as when a Windows 10 user uploads a file to a web server that's running the Linux operating system. Using newLine() instead of a literal (such as '\n') makes your program more user-friendly across different platforms.

The close() method is called to close the buffered character stream and make sure that all buffered data is sent to the stream's destination.

Files and Paths

In all the examples thus far, a string has been used to refer to the file that's involved in a stream operation. This often is sufficient for a program that uses files and streams, but if you want to copy or rename files or handle other tasks, you can use a Path object from the java.nio.file package.

Path represents a file or folder reference. It is an improvement on the File class in the java.io package. The following statement gets a path matching the specified string:

Click here to view code image

```
Path source = FileSystems.getDefault().getPath("essay.txt");
```

This is a two-step process. First, a class method of the FileSystems class is called. The getDefault() method returns a FileSystem object that represents the computer's way of storing files. Both of these classes also are in the java.nio.file package.

As soon as you have that `FileSystem` object, its `getPath(String)` method returns a `Path` object matching that specified file or folder reference.

A `File` object can be created from a `Path` by calling the `toFile()` method of the latter class, as in this statement:

Click here to view code image

```
File sourceFile = source.toFile();
```

A `Path` object can be created from a `File` by calling its `toPath()` method.

You can call several class methods of the `Files` class in the `java.nio.file` package when working with files.

The `move (Path, Path)` class method renames a file from the first path argument to the second.

The `delete(Path)` class method deletes that file.

Just like any file-handling operation, these methods must be handled with care to avoid deleting the wrong files and folders or wiping out data.

These methods throw a `SecurityException` if the program does not have the security to perform the file operation in question, a `NoSuchFileException` if the paths do not exist, and an `IOException` for other IO errors. If you try to delete a folder that is not empty, a `NoSuchFileException` exception occurs. Therefore, these exceptions need to be dealt with through a `try-catch` block or a `throws` clause in a method declaration.

The `AllCapsDemo` application shown in Listing 15.8 converts all the text in a file to uppercase characters. The file is pulled in using a buffered input stream, and one character is read at a time. After the character is converted to uppercase, it is sent to a temporary file using a buffered output stream. `File` objects are used instead of strings to indicate the files involved, which makes it possible to rename and delete files as needed. In NetBeans, create an empty Java file called `AllCapsDemo` in the `com.java21days` package.

LISTING 15.8 The Full Text of `AllCapsDemo.java`

Click here to view code image

```
1: package com.java21days;
2:
```

```
 3: import java.io.*;
 4: import java.nio.file.*;
 5:
 6: public class AllCapsDemo {
 7:     public static void main(String[] arguments) {
 8:         if (arguments.length < 1) {
 9:             System.out.println("You must specify a
filename");
10:             System.exit(-1);
11:         }
12:         AllCaps cap = new AllCaps(arguments[0]);
13:         cap.convert();
14:     }
15: }
16:
17: class AllCaps {
18:     String sourceName;
19:
20:     AllCaps(String sourceArg) {
21:         sourceName = sourceArg;
22:     }
23:
24:     void convert() {
25:         try {
26:             // Create file objects
27:             FileSystem fs = FileSystems.getDefault();
28:             Path source = fs.getPath(sourceName);
29:             Path temp = fs.getPath("tmp_" + sourceName);
30:
31:             // Create input stream
32:             FileReader fr = new FileReader(source.toFile());
33:             BufferedReader in = new BufferedReader(fr);
34:
35:             // Create output stream
36:             FileWriter fw = new FileWriter(temp.toFile());
37:             BufferedWriter out = new
38:                 BufferedWriter(fw);
39:
40:             boolean eof = false;
41:             int inChar;
42:             do {
43:                 inChar = in.read();
44:                 if (inChar != -1) {
45:                     char outChar = Character.toUpperCase(
46:                         (char) inChar);
47:                     out.write(outChar);
48:                 } else
49:                     eof = true;
50:             } while (!eof);
51:             in.close();
52:             out.close();
```

```
53:
54:             Files.delete(source);
55:             Files.move(temp, source);
56:         } catch (IOException|SecurityException se) {
57:             System.out.println("Error -- " + se.toString());
58:         }
59:     }
60: }
```

Before running the program, you need a text file that can be converted to all capital letters. One option is to make a copy of `AllCapsDemo.java` and give it a name like `TempFile.java`. This file should be stored in the root project folder in NetBeans and specified as a command-line argument.

This program does not produce any output. Load the converted file into a text editor to see the results of the application.

Summary

Today you learned how to work with streams in two directions: pulling data into a program over an input stream and sending data from a program using an output stream.

You used character streams to handle text and byte streams for any other kind of data. Filters were associated with streams to alter how information was delivered through a stream, or to alter the information itself.

In addition to these classes, `java.io` offers other types of streams you might want to explore. Piped streams are useful when communicating data among different threads, and byte array streams can connect programs to a computer's memory.

Because the stream classes in Java are so closely coordinated, you already possess most of the knowledge you need to use these other types of streams. The constructors, read methods, and write methods are largely identical.

Streams are a powerful way to extend the functionality of your Java programs because they offer a connection to any kind of data you might want to work with.

Tomorrow, you will use streams to read and write Java objects.

Q&A

Q The C program that I use creates a file of integers and other data. Can I read this using a Java program?

A You can, but one thing you have to consider is whether your C program represents integers in the same manner that a Java program represents them. As you might recall, all data can be represented as an individual byte or a series of bytes. An integer is represented in Java using 4 bytes arranged in what is called big-endian order. You can determine the integer value by combining the bytes from left to right. A C program implemented on an Intel PC is likely to represent integers in little-endian order, which means that the bytes must be arranged from right to left to determine the result. You might have to learn about advanced techniques, such as bit shifting, to use a data file created with a programming language other than Java.

Q Can relative paths be used when specifying the name of a file in Java?

A Relative paths are determined according to the current user folder, which is stored in the system properties `user.dir`. You can find out the full path to this folder by using the `System` class in the main `java.lang` package, which does not need to be imported.

Call the `System` class `getProperty(String)` method with the name of the property to retrieve, as in this example:

Click here to view code image

```
String userFolder = System.getProperty("user.dir");
```

The method returns the path as a string.

Q The `FileWriter` class has a `write(int)` method that's used to send a character to a file. Shouldn't this be `write(char)`?

A The `char` and `int` data types are interchangeable in many ways; you can use an `int` in a method that expects a `char`, and vice versa. This is possible because each character is represented by a numeric code that is an integer value. When you call the `write()` method with an `int`, it outputs the character associated with that integer value. When calling the `write()` method, you can cast an `int` value to a `char` to ensure that it's being used as you intended.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What happens when you create a `FileOutputStream` using a reference to an existing file?

   A. An exception is thrown.

   B. The data you write to the stream is appended to the existing file.

   C. The existing file is replaced with the data you write to the stream.

2. What two primitive types are interchangeable when you're working with streams?

   A. `byte` and `boolean`

   B. `char` and `int`

   C. `byte` and `char`

3. In Java, what is the maximum value of a `byte` variable and the maximum value of an unsigned byte in a stream?

   A. Both are 255.

   B. Both are 127.

   C. 127 for a `byte` variable and 255 for an unsigned byte

Answers

1. C. That's one of the things to look out for when using output streams; you can easily wipe out existing files. Constructors can use a Boolean value to append data to a file instead of replacing the entire thing.
2. B. Because Java represents a `char` internally as an integer value, you often can use the two interchangeably in method calls and other statements.
3. C. The `byte` primitive data type has values ranging from –128 to 127, whereas an unsigned byte can range from 0 to 255.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
import java.io.*;

public class Unknown {
    public static void main(String[] arguments) {
        String command = "";
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
        try {
            command = br.readLine();
        }
        catch (IOException e) { }
    }
}
```

Will this program successfully store a line of console input in the `String` object named `command`?

A. Yes.

B. No, because a buffered input stream is required to read console input.

C. No, because it won't compile successfully.

D. No, because it reads more than one line of console input.

The answer is available on the book's website at www.java21days.com. Visit the Day 15 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Write a modified version of the HexReader program from Day 7, that reads two-digit hexadecimal sequences from a text file and displays their decimal equivalents.
2. Write a program that reads a file to determine the number of bytes it contains and then overwrites all those bytes with 0s. (For obvious

reasons, don't test this program on any file you intend to keep, because the file's data will be wiped out.)

Exercise solutions are offered on the book's website at www.java21days.com.

Day 16. Using Inner Classes and Closures

Each new version of the Java language takes it further from its humble origins in 1995. When it was first released, Java had only 250 classes in the Java Class Library and was primarily used to put interactive programs on Web pages. These applets, as they were dubbed, brought something new to the Web and inspired several hundred thousand programmers to learn the new language.

Because the language was well-designed and offered some features that made it a worthy rival to C++ and other choices for software development, Java quickly outgrew its original focus to become a general-purpose programming language. Today it is the most widely implemented, popular language in the world.

There are millions of Java coders today putting its classes on several billion devices as the language turns 20. Each new release embraces new capabilities that bring sophisticated new methodologies to Java that are eagerly anticipated by programmers.

Java 8 offers a new feature that may be the most requested ever: closures.

Closures, also called lambda expressions, make it possible in Java to employ a methodology called functional programming.

Today you will learn about closures after an introduction to two parts of the language that are prerequisites to their use: inner classes and anonymous inner classes.

Inner Classes

When you create a class in Java, you must define its attributes and behavior. The attributes are the class and instance variables that hold its data, and the behavior is the methods that use that data to perform tasks.

A class also can contain a third element that combines both attributes and behavior—an inner class.

Inner classes are like helper classes, but they are defined inside the class they were created to help. Because a Java program can have as many classes as you think are necessary, you might be questioning the point of inner classes. A `Scheduler` class that manages work schedules at your

restaurant could have an `Employee` helper class for each worker and a `Day` helper class for each weekday the business is open.

Though some of the purpose of an inner class can be accomplished with a helper class, as you learn more about them you will encounter situations where they're better suited to a particular project.

Java includes inner classes for several reasons.

If a class is used by only one other class, it's a good idea to define it inside that class. That keeps the code in one place and makes clear the relationship between the classes.

An inner class can access private methods and variables of its enclosing class that a helper class could not access. This is possible for the same reason that a method in a class can access private variables of that class.

Note

Rules governing the scope of an inner class closely match those governing variables. An inner class's name is not visible outside its scope, except in a fully qualified name (the enclosing class name followed by a period and inner class name). This helps in structuring classes within a package. The code for an inner class can use simple names from enclosing scopes, including class and member variables of enclosing classes, as well as local variables of enclosing blocks.

To create an inner class, use the `class` keyword and a class declaration like any other class, but place it inside the containing class. An inner class usually is put in the same place that class and instance variables are defined.

Here's an inner class called `InnerHello` in a class called `Hello`:

Click here to view code image

```
public class Hello {

    class InnerHello {
        InnerHello() {
            System.out.println(
                "The method call is coming from inside the
class!"
            );
        }
    }
```

```
        public Hello() {
            // empty constructor
        }

        public static void main(String[] arguments) {
            Hello program = new Hello();
            Hello.InnerHello inner = program.new InnerHello();
        }
    }
```

The inner class is defined just like any other class, except for its position: It is placed within the { and } brackets of another class.

Creating an object of an inner class requires an object of the outer class. The `new` operator is called on the object, as in this statement from the preceding example:

Click here to view code image

```
Hello.InnerHello inner = program.new InnerHello();
```

Look at both halves of this assignment statement to learn how the object of the inner class is created.

On the left, the name of the inner class consists of the name of the outer class, a period character (“.”) and the inner class name. So `Hello.InnerHello` is the name.

On the right, `program` refers to the `Hello` object. The reference to `program` is followed by a period, the `new` operator, and the inner class constructor `InnerHello()`.

The day’s first project rewrites the ComicBook application from Day 8, “Data Structures,” to use an inner class. That project managed a comic book collection with a main class called `ComicBooks` and a helper class called `Comic` for each comic in a collection. Both were defined in the same source code file, but as separate classes. The compiler turned them into the bytecode files `ComicBooks.class` and `Comic.class`.

This time around, there’s a `ComicBox` class for the collection and an `InnerComic` inner class.

The `ComicBox` application is shown in Listing 16.1. Create a new empty Java file in the `com.java21days` package for this project called `ComicBox.java`.

LISTING 16.1 The Full Text of `ComicBox.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class ComicBox {
 6:     class InnerComic {
 7:         String title;
 8:         String issueNumber;
 9:         String condition;
10:         float basePrice;
11:         float price;
12:
13:         InnerComic(String inTitle, String inIssueNumber,
14:             String inCondition, float inBasePrice) {
15:
16:             title = inTitle;
17:             issueNumber = inIssueNumber;
18:             condition = inCondition;
19:             basePrice = inBasePrice;
20:         }
21:
22:         void setPrice(float factor) {
23:             price = basePrice * factor;
24:         }
25:     }
26:
27:     public ComicBox() {
28:         HashMap<String, Float> quality = new HashMap<>();
29:         float price1 = 3.00F;
30:         quality.put("mint", price1);
31:         float price2 = 2.00F;
32:         quality.put("near mint", price2);
33:         float price3 = 1.50F;
34:         quality.put("very fine", price3);
35:         float price4 = 1.00F;
36:         quality.put("fine", price4);
37:         float price5 = 0.50F;
38:         quality.put("good", price5);
39:         float price6 = 0.25F;
40:         quality.put("poor", price6);
41:         InnerComic[] comix = new InnerComic[3];
42:         comix[0] = new InnerComic("Amazing Spider-Man",
"1A",
43:             "very fine", 12_000.00F);
44:         comix[0].setPrice(quality.get(comix[0].condition));
```

```
45:          comix[1] = new InnerComic("Incredible Hulk", "181",
46:              "near mint", 680.00F);
47:          comix[1].setPrice(quality.get(comix[1].condition));
48:          comix[2] = new InnerComic("Cerebus", "1A", "good",
190.00F);
49:          comix[2].setPrice(quality.get(comix[2].condition));
50:          for (InnerComic comix1 : comix) {
51:              System.out.println("Title: " + comix1.title);
52:              System.out.println("Issue: " +
comix1.issueNumber);
53:              System.out.println("Condition: " +
comix1.condition);
54:              System.out.println("Price: $" + comix1.price +
"\n");
55:          }
56:      }
57:
58:      public static void main(String[] arguments) {
59:          new ComicBox();
60:      }
61: }
```

The inner class, which is defined in Lines 6–25, has a constructor that creates a comic book using a title, issue number, condition, and base price. There's also a `setPrice()` method in Lines 22–24.

The `ComicBox` class uses this inner class in Line 41, creating an array that holds three `InnerComic` objects. The inner class is referred to as `InnerComic`, the same as if it was a helper class.

You also could have referred to the inner class using its full name, which includes the name of its enclosing class:

Click here to view code image

```
ComicBox.InnerComic[] comix = new ComicBox.InnerComic[3];
```

The output of the application is displayed in Figure 16.1.

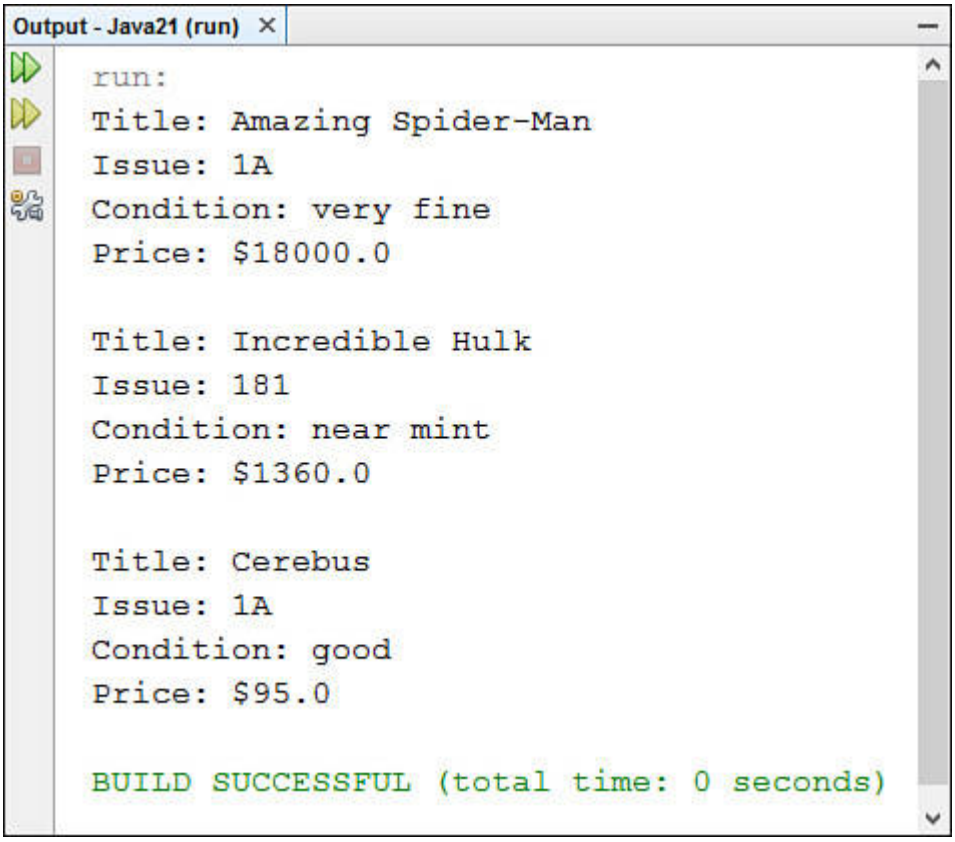


FIGURE 16.1 Using inner classes to collect comic books.

Anonymous Inner Classes

Often in Java programming you need to create an object in one statement that never will be referred to again. There's a special type of inner class well-suited to this purpose: an anonymous inner class. This is a class that has no name and is declared and created in the same statement.

To use an anonymous inner class, you take a place where you'd refer to an object's variable and replace it with the `new` keyword, a call to a constructor and the class definition inside { and } characters.

The purpose will make more sense when you see how it replaces code that doesn't use one of these classes.

The following code creates a thread and starts it without using anonymous inner classes:

Click here to view code image

```
ThreadClass task = new ThreadClass();
Thread runner = new Thread(task);
runner.start();
```

For this example, assume the `task` object implements the `Runnable` interface to be run as a thread. Assume as well that the code in `ThreadClass` is simple and the class needs to be used only once.

In this situation, it's efficient to get rid of `ThreadClass` and put its code inside an anonymous inner class. This code rewrite does exactly that:

Click here to view code image

```
Thread runner = new Thread(new Runnable()  {
    public void run() {
        // thread does its work here
    }
});
runner.start();
```

The anonymous inner class has replaced the reference to `task` with the following code:

Click here to view code image

```
new Runnable()  {
    public void run() {
        // thread does its work here
    }
)
```

In Java, calling the `new` operator is an expression that returns an object. So putting this code inside the `Thread()` constructor returns an unnamed object that implements the `Runnable` interface and overrides the `run()` method. The statements inside that method do the work that has been put in its own thread.

For a deeper look at this concept, the next project will be a full demonstration of how anonymous inner classes are created and why they're so useful.

On Day 12, "Responding to User Input," you learned about how to monitor user input in a Swing application by using interfaces called event listeners. When an application must monitor a particular type of input, such as a user clicking a button, moving a mouse, or typing keys on the keyboard, it must have a class that implements the listener interface for that input. These classes are in the package `java.awt.event`.

User clicks are monitored by `KeyListener`, for instance.

One event listener that was not covered is `WindowListener`, which tracks the different ways a user can interact with a window.

There are methods in the `WindowListener` interface for when a window has been opened and closed, as well as when it has become the focus or lost the focus.

A class that implements the interface must implement 10 methods: `windowActivated()`, `windowClosed()`, `windowClosing()`, `windowDeactivated()`, `windowDeiconified()`, `windowGainedFocus()`, `windowIconified()`, `windowLostFocus()`, `windowOpened()`, and `windowStateChanged()`.

That's a lot of methods to implement, especially if you have only one or two possible window interactions that your class is interested in. A frame that only monitors when a window opened would have code that looked something like this:

Click here to view code image

```
public void windowOpened(WindowEvent event) {
    Window pane = event.getWindow();
    pane.setBackground(Color.CYAN);
}

public void windowClosed(WindowEvent event) {
    // do nothing
}

public void windowActivated(WindowEvent event) {
    // do nothing
}

public void windowDeactivated(WindowEvent event) {
    // do nothing
}
```

That's just the part of the window event code required. There are another six do-nothing methods that must be present in a class that implements the `WindowListener` interface.

After all 10 methods are implemented in a frame, the frame can add a listener to monitor window events:

```
addWindowListener(this);
```

There's a better way to create the listener and add it to the frame: Use a subclass of the `WindowAdapter` class.

The `WindowAdapter` class implements the `WindowListener` interface as 10 methods that each do nothing. There are several adapter classes in the `java.awt.event` that simplify the process of listening to a particular event. You can create a subclass of the adapter class that overrides the method (or methods) only where something needs to happen.

Here's code for a window listener that uses `WindowAdapter` and only monitors the `windowClosing()` event:

Click here to view code image

```
public class WindowCloseListener extends WindowAdapter {
    JFrame frame;
    boolean done;

    public WindowCloseListener(JFrame inFrame) {
        this.frame = inFrame;
    }

    public void windowClosing(WindowEvent event) {
        // user has tried to close window
       if (frame.done) {
            // allow it
            frame.dispose();
            System.exit(0);
        }
    }
}
```

Calling a window's `dispose()` method closes it. This code waits for a user to close a frame and does it only when the Boolean variable `done` equals `true`. That variable is an instance variable of a frame in another class (the one that created the listener).

In that frame, the frame's default behavior must be set to ignore attempts to close the window:

Click here to view code image

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

Also, the frame creates the listener object and makes a listener:

Click here to view code image

```
WindowCloseListener closer = new WindowCloseListener();
addWindowListener(closer);
```

An object of the helper class `WindowCloseListener` is assigned to a variable and set to monitor window events.

This approach to monitoring one window event requires four steps:

1. Create a subclass of `WindowAdapter`.
2. Implement the window closing method in that class.
3. Create a constructor in that class with the frame that needs the class as an argument.
4. Store that frame in an instance variable.

The constructor and instance variable are needed to link the two classes. The adapter must be able to access the frame's `done` variable.

A simpler approach can be accomplished in the frame's class through the use of an anonymous inner class:

Click here to view code image

```
setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
addWindowListener(new WindowAdapter() {
    // user has tried to close window
    if (frame.done) {
        // allow it
        frame.dispose();
        System.exit(0);
    }
});
```

The listener is created anonymously by calling `new WindowAdapter()` with a definition of the class. The class overrides the `windowClosing()` method so that when a user closes a window, an action can be taken.

This anonymous inner class can do something that a separate helper class could not do—access the `frame` instance variable. Inner classes are able to access the methods and variables of their enclosing class, just like instance variables and methods.

Note

There are other adapter classes in the `java.awt.event` packages that make it convenient to implement other listeners. The `KeyAdapter` class has empty methods for keyboard events, `MouseAdapter` for mouse events, and `FocusAdapter` for keyboard focus events.

During Day 10, "Building a Swing Interface," you created a ProgressMonitor application that used a slider as a progress bar. The next project today will enhance that code to prevent the program's main window from being closed if the progress bar has not reached 100 percent.

In NetBeans, create a new empty Java file named `ProgressMonitor2` in the class `com.java21days`, then enter the text of Listing 16.2 into that file. Save your work when you're done.

LISTING 16.2 The Full Text of `ProgressMonitor2.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class ProgressMonitor2 extends JFrame {
 8:     JProgressBar current;
 9:     int num = 0;
10:     boolean done = false;
11:
12:     public ProgressMonitor2() {
13:         super("Progress Monitor 2");
14:         setLookAndFeel();
15:         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE)
;
16:         addWindowListener(new WindowAdapter() {
17:             @Override
18:             public void windowClosing(WindowEvent event) {
19:                 // user has tried to close window
20:                 if (done) {
21:                     // allow it
22:                     dispose();
```

```
23:                     System.exit(0);
24:                 }
25:             }
26:         });
27:         setSize(400, 100);
28:         setLayout(new FlowLayout());
29:         current = new JProgressBar(0, 2000);
30:         current.setValue(0);
31:         current.setStringPainted(true);
32:         current.setPreferredSize(new Dimension(360, 48));
33:         add(current);
34:         setVisible(true);
35:         iterate();
36:     }
37:
38:     public final void iterate() {
39:         while (num < 2000) {
40:             current.setValue(num);
41:             try {
42:                 Thread.sleep(1000);
43:             } catch (InterruptedException e) { }
44:             num += 95;
45:         }
46:         done = true;
47:     }
48:
49:     private void setLookAndFeel() {
50:         try {
51:             UIManager.setLookAndFeel(
52:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
53:             );
54:             SwingUtilities.updateComponentTreeUI(this);
55:         } catch (Exception e) {
56:             System.err.println("Couldn't use the system "
57:                 + "look and feel: " + e);
58:         }
59:     }
60:
61:     public static void main(String[] arguments) {
62:         new ProgressMonitor2();
63:     }
64: }
```

The anonymous inner class is created and used in Lines 16–26. It monitors window input using `windowClosing()`, the only method in the `WindowListener` interface the application needs to check, and makes sure the `done` instance variable equals `true` before closing the window.

See Figure 16.2 for the program's output.

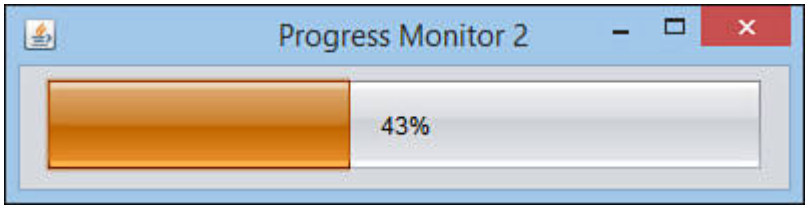


FIGURE 16.2 Stopping a program from closing until its work is done.

Anonymous inner classes cannot have a constructor, making them more limited than other inner classes and helper classes.

Anonymous inner classes are a bit more complex than other aspects of the Java language. They look odd in the source code of a program, and getting the punctuation right is tricky as you begin working with them. When you've added them to your skill set, you will find that they are a powerful, flexible, and concise way to get things done.

Closures

Java 8 adds the most highly requested language feature in years: closures. Programmers of other languages that offer them, such as Scala and Smalltalk, have clamored for them during the development of the past several releases.

Closures, also called lambda expressions, enable an object from a class with only a single method to be created with an `->` operator, provided that other conditions are met.

This statement is an example:

Click here to view code image

```
Runnable runner = () -> { System.out.println("Eureka!"); };
```

This code creates an object that implements the `Runnable` interface with a `run()` method equivalent to the following code:

Click here to view code image

```
public void run() {
    System.out.println("Eureka!");
}
```

In a closure, the statement to the right of the `->` arrow operator defines the method that implements the interface.

This is possible only when the interface has a single method to implement, as `Runnable` does with only the `run()` method. When an interface in Java has one method, it's now called a functional interface.

As you might have spotted, a closure also has something unusual to the left of the arrow operator. In the `Runnable` example, it's an empty set of parentheses.

This part of the expression holds the arguments to send the method of the functional interface. The `run()` takes no arguments in the `Runnable` interface, so no arguments are required in that expression.

Take a look at another example of a closure that does have something inside the parentheses on the left side of the expression:

Click here to view code image

```
ActionListener listen = (ActionEvent act) -> {
    System.out.println(act.getSource());
};
```

The closure provides an implementation of the only method in the `ActionListener` interface, `actionPerformed()`. That method takes one argument, an `ActionEvent` object. `ActionListener` is in the `java.awt.event` package.

Here's the nonclosure way to implement the same functionality:

Click here to view code image

```
public void actionPerformed(ActionEvent act) {
    System.out.println(act.getSource());
}
```

The `ActionListener` interface handles action events such as a user's button click or menu item selection. The only method in the functional interface is `actionPerformed(`*`ActionEvent`*`)`. The argument contains the user action that triggered the event.

The right half of the closure defines the `actionPerformed()` method as a statement that displays information about the interface component where the event happened. The left half makes an `ActionEvent` object the argument to the method.

This object, `act`, is used inside the body of the method. In the closure, the left-half reference to `act` appears to be outside the scope of the right-half

method implementation. Closures allow code to refer to variables of another method outside the scope of those variables.

Like anonymous inner classes, closures have the effect of making code shorter. A single expression creates an object and implements an interface.

Closures can make code even shorter through Java's support for target typing.

In a closure, it's possible to infer the class of the argument (or arguments) sent to the method. In the `ActionListener` example, the functional interface has a method with an `ActionEvent` object as its only argument. For this reason, the name of the class can be omitted.

Here's a simplified version of the closure taking this into account:

Click here to view code image

```
ActionListener listen = (act) -> {
    System.out.println(act.getSource());
}
```

Today's final two programs illustrate the difference that closures bring to Java.

The `CursorMayhem` application in Listing 16.3 is a Swing program that displays three buttons in a panel that change the program's cursor.

Cursors have not been covered up to this point, but they're simple to use. They're represented by the `Cursor` class in the `java.awt` package and can be changed by calling a container's `setCursor(`*`Cursor`*`)` method.

The type of cursor is determined by class variables of the class. The following statements create a panel and set its cursor to the one used in text boxes:

Click here to view code image

```
JPanel panel = new JPanel();
panel.setCursor(new Cursor(Cursor.TEXT_CURSOR));
```

This statement sets the cursor back to the default:

Click here to view code image

```
panel.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
```

This application will be implemented two different ways. The first version uses an anonymous inner class, not a closure, to monitor user clicks on the three buttons. Create a new program in NetBeans with the name

`CursorMayhem` and the package `com.java21days`, then type in the text in Listing 16.3.

LISTING 16.3 The Full Text of `CursorMayhem.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class CursorMayhem extends JFrame {
 8:     JButton harry, wade, hansel;
 9:
10:     public CursorMayhem() {
11:         super("Choose a Cursor");
12:         setLookAndFeel();
13:         setSize(400, 80);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         setLayout(new FlowLayout());
16:         harry = new JButton("Crosshair");
17:         add(harry);
18:         wade = new JButton("Wait");
19:         add(wade);
20:         hansel = new JButton("Hand");
21:         add(hansel);
22:         // begin anonymous inner class
23:         ActionListener act = new ActionListener() {
24:             public void actionPerformed(ActionEvent event) {
25:                 if (event.getSource() == harry) {
26:                     setCursor(new
Cursor(Cursor.CROSSHAIR_CURSOR));
27:                 }
28:                 if (event.getSource() == wade) {
29:                     setCursor(new
Cursor(Cursor.WAIT_CURSOR));
30:                 }
31:                 if (event.getSource() == hansel) {
32:                     setCursor(new
Cursor(Cursor.HAND_CURSOR));
33:                 }
34:             }
35:         };
36:         // end anonymous inner class
37:         harry.addActionListener(act);
38:         wade.addActionListener(act);
```

```
39:          hansel.addActionListener(act);
40:          setVisible(true);
41:      }
42:
43:      private void setLookAndFeel() {
44:          try {
45:              UIManager.setLookAndFeel(
46:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
47:              );
48:          } catch (Exception exc) {
49:              System.err.println("Look and feel error: " +
exc);
50:          }
51:      }
52:
53:      public static void main(String[] arguments) {
54:          new CursorMayhem();
55:      }
56: }
```

This program is shown running in Figure 16.3.

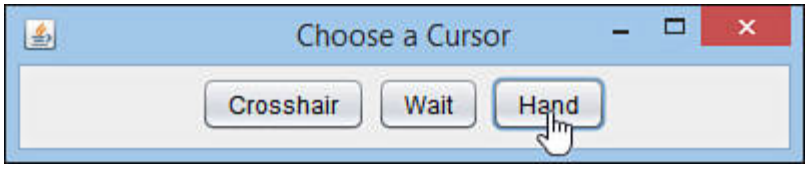


FIGURE 16.3 Monitoring action events with an anonymous inner class.

Lines 23–35 of the application define an event listener for the `CursorMayhem` class using an anonymous inner class. This nameless object contains an implementation of the only method in the `ActionListener` interface: `actionPerformed(`*`ActionEvent`*`)`.

In the method, the frame's cursor is changed by calling its `setCursor()` method. Anonymous inner classes have access to the methods and instance variables of their enclosing class. A separate helper class would lack that access.

As you are running the app, move your cursor over the title bar that reads "Choose a Cursor." It changes from the current cursor to the default. Move it back over the pane and it becomes the cursor that you selected again.

Now take a look at the `ClosureMayhem` application in Listing 16.4.

LISTING 16.4 The Full Text of `ClosureMayhem.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class ClosureMayhem extends JFrame {
 8:     JButton harry, wade, hansel;
 9:
10:     public ClosureMayhem() {
11:         super("Choose a Cursor");
12:         setLookAndFeel();
13:         setSize(400, 80);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         setLayout(new FlowLayout());
16:         harry = new JButton("Crosshair");
17:         add(harry);
18:         wade = new JButton("Wait");
19:         add(wade);
20:         hansel = new JButton("Hand");
21:         add(hansel);
22:         // begin closure
23:         ActionListener act = (event) -> {
24:             if (event.getSource() == harry) {
25:                 setCursor(new
Cursor(Cursor.CROSSHAIR_CURSOR));
26:             }
27:             if (event.getSource() == wade) {
28:                 setCursor(new Cursor(Cursor.WAIT_CURSOR));
29:             }
30:             if (event.getSource() == hansel) {
31:                 setCursor(new Cursor(Cursor.HAND_CURSOR));
32:             }
33:         };
34:         // end closure
35:         harry.addActionListener(act);
36:         wade.addActionListener(act);
37:         hansel.addActionListener(act);
38:         setVisible(true);
39:     }
40:
41:     private void setLookAndFeel() {
42:         try {
43:             UIManager.setLookAndFeel(
44:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAn
dFeel"
45:             );
46:         } catch (Exception exc) {
```

```
47:             System.err.println("Look and feel error: " +
exc);
48:         }
49:     }
51:
52:     public static void main(String[] arguments) {
53:         new ClosureMayhem();
54:     }
55: }
```

The `ClosureMayhem` application implements the action listener in lines 23–33. Everything else is the same as in `CursorMayhem`, except for the lines that refer to the class name.

In `ClosureMayhem`, you don't need to know the name of the method in the `ActionListener` interface to use it in the program. You also don't need to specify the class of the `ActionEvent` that is the method's only argument.

Closures support functional programming, a methodology for software design that has before Java 8 been unavailable in the language.

With the basic syntax of closures and two common ways they can be employed in programs, you can begin to exploit this feature. At this point you should be able to recognize closures, write statements involving the arrow operator =>, and use it to create an object for any single-method interface.

These single-method interfaces also are called functional interfaces.

Summary

Like generics, inner classes, anonymous inner classes, and closures are among the most sophisticated aspects of Java. These advanced features are most important to programmers who have been working in the language long enough to develop a high level of experience in its use. An expert coder knows how to employ these features to do more in less code.

Before you reach a point where you are comfortable writing closures, you ought to be able to benefit from inner classes and anonymous inner classes.

A non-anonymous inner class takes the form of a helper class, situated inside a class instead of on its own. The class is defined with the instance variables, class variables, instance methods, and class methods that make up the behavior and attributes of the enclosing class. Because it's defined inside

that class, the inner class can read and write its private variables and methods of the class.

An anonymous inner class is created without needing a variable, which makes sense for an object that will be used only once in a program. These classes often are used when a Swing user interface component needs an event listener to monitor user input.

Closures are deceptively similar in appearance, requiring only the new -> arrow operator to create, but offer an enormous enhancement to a Java programmer's capabilities.

Q&A

Q Is it necessary to use anonymous inner classes?

A Whenever you can get something done without a feature of the Java language, you don't have to use that feature. Programmers generally can accomplish a task in a program in a bunch of different ways. Though any sophisticated new technique is likely to work in fewer lines of code or offer other advantages, there's no penalty in Java for doing something in more statements.

Generally, what matters is that your program works, not how many lines it took to make that happen.

With that disclaimer, you should become conversant in features such as anonymous inner classes anyway. They are something you are going to find in Java code. Experienced programmers use inner classes, anonymous inner classes, and closures. Knowing what they are will help you understand what someone else's code is doing.

Q Why are closures also called lambda expressions?

A The term "lambda" comes from the system of math logic called lambda calculus, where the Greek letter lambda represents an anonymous function. The choice of this letter was arbitrary.

Lambda calculus has proven to be extremely useful in math, computation theory, and computer programming.

Closures are available in many programming languages today in addition to Java. They include JavaScript, Python, C#, Scala, Smalltalk, and Haskell.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What can an inner class access that a separate helper class could not?
   - **A.** Anonymous inner classes
   - **B.** `private` variables of another class
   - **C.** Threads
2. What makes a Java interface qualified to be called a functional interface?
   - **A.** The number of methods in that interface
   - **B.** The arrow operator
   - **C.** Any interface can be functional.
3. What does an adapter class make easier?
   - **A.** The use of closures
   - **B.** Arranging Swing user interface components
   - **C.** Implementing an event listener

Answers

1. B. An inner class can access the `private` variables and methods of its enclosing class.
2. A. An interface that defines only one method is a functional interface.
3. C. An adapter implements all the methods in an event listener interface so you can subclass the adapter and override only the method or methods that are useful.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class ClassType {
    public static void main(String[] arguments) {
        Class c = String.class;
        try {
            Object o = c.newInstance();
            if (o instanceof String) {
                System.out.println("True");
            } else {
                System.out.println("False");
            }
        } catch (Exception e) {
            System.out.println("Error");
        }
    }
}
```

What will be the output of this application?

A. `true`

B. `false`

C. Error

D. The program will not compile.

The answer is available on the book's website at www.java21days.com. Visit the Day 16 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a new version of the `DiceRoller` program from Day 14, "Developing Swing Applications," that makes `DiceWorker` an inner class.
2. Extend your new `DiceRoller` program to monitor action events with a closure.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 17. Communicating Across the Internet

Java was developed initially as a language that would control a network of interactive consumer devices. Connecting machines was one of the main purposes of the language when it was designed, and that remains true today.

The `java.net` package makes it possible to communicate over a network, providing cross-platform abstractions to make connections, transfer files using common web protocols, and create sockets.

Used in conjunction with input and output streams, reading and writing files over the network becomes as easy as reading or writing files on disk.

The `java.nio` package expands Java's input and output classes.

Today you write networking Java programs that do each of the following:

- Load a document over the Web
- Mimic a popular Internet service
- Serve information to clients

Networking in Java

Networking allows different computers to make connections with each other and exchange information. In Java, basic networking is supported by classes in the `java.net` package, including support for connecting and retrieving files through Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP), as well as working at a lower level with sockets.

You can communicate with systems on the Net in three simple ways:

- Load a web page and any other resource with a uniform resource locator (URL).
- Use the socket classes, `Socket` and `ServerSocket`, which open standard socket connections to hosts and read to and write from those connections.
- Call `getInputStream()`, a method that opens a connection to a URL and can extract data from that connection.

Opening a Stream Over the Net

As you learned during Day 15, "Working with Input and Output," you can pull information through a stream into your Java programs in several ways. The classes and methods you choose depend on the format of the information and what you want to do with it.

One of the resources you can reach from your Java programs is a text document on the Web, whether it's an HTML file, XML file, or some other kind of plain-text document.

You can use a four-step process to load a text document off the Web and read it line by line:

1. Create a `URL` object that represents the resource's web address.
2. Create an `HttpURLConnection` object that can load the URL and make a connection to the site hosting it.
3. Use the `getContent()` method of that `HttpURLConnection` object to create an `InputStreamReader` that can read a stream of data from the URL.
4. Use that input stream reader to create a `BufferedReader` object that can efficiently read characters from an input stream.

Much interaction occurs between the web document and your Java program. The URL is used to set up a URL connection, which is used to set up an input stream reader, which is used to set up a buffered input stream reader. The need to deal with any exceptions that occur along the way adds more complexity to the process.

Before you can load anything, you must create a new instance of the class `URL` that represents the address of the resource you want to load. URL is an acronym for *uniform resource locator*, and it refers to the unique address of any document or other resource accessible on the Internet.

`URL` is part of the `java.net` package, so you must import the package or refer to the class by its full name in your programs.

To create a new `URL` object, use one of four constructors:

- `URL(String)` creates a `URL` object from a full web address such as "http://www.java21days.com" or "ftp://ftp.freebsd.org".

- `URL(URL, String)` creates a `URL` object with a base address provided by the specified `URL` and a relative path provided by the `String`.
- `URL(String, String, int, String)` creates a new `URL` object from a protocol (such as "`http`" or "`ftp`"), hostname (such as "`www.cnn.com`" or "`web.archive.org`"), port number (80 for HTTP), and filename or pathname.
- `URL(String, String, String)` is the same as the previous constructor minus the port number.

When you use the `URL(String)` constructor, you must deal with `MalformedURLException` exceptions, which are thrown if the string does not appear to be a valid URL. These objects can be handled in a `try-catch` block:

Click here to view code image

```
try {
    URL load = new URL("http://www.samspublishing.com");
} catch (MalformedURLException e) {
    System.out.println("Bad URL");
}
```

The WebReader application, shown in Listing 17.1, uses the four-step technique to open a connection to a website and read a text document from it. When the document is fully loaded, it is displayed in a text area. Create this class in NetBeans in the `com.java21days` package.

LISTING 17.1 The Full Text of `WebReader.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4: import java.net.*;
 5: import java.io.*;
 6:
 7: public class WebReader extends JFrame {
 8:     JTextArea box = new JTextArea("Getting data ...");
 9:
10:     public WebReader() {
11:         super("Get File Application");
12:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
13:         setSize(600, 300);
14:         JScrollPane pane = new JScrollPane(box);
15:         add(pane);
16:         setVisible(true);
17:     }
18:
19:     void getData(String address) throws MalformedURLException
{
20:         setTitle(address);
21:         URL page = new URL(address);
22:         StringBuilder text = new StringBuilder();
23:         try {
24:             HttpURLConnection conn = (HttpURLConnection)
25:                 page.openConnection();
26:             conn.connect();
27:             InputStreamReader in = new InputStreamReader(
28:                 (InputStream) conn.getContent());
29:             BufferedReader buff = new BufferedReader(in);
30:             box.setText("Getting data ...");
31:             String line;
32:             do {
33:                 line = buff.readLine();
34:                 text.append(line);
35:                 text.append("\n");
36:             } while (line != null);
37:             box.setText(text.toString());
38:         } catch (IOException ioe) {
39:             System.out.println("IO Error:" +
ioe.getMessage());
40:         }
41:     }
42:
43:     public static void main(String[] arguments) {
44:         if (arguments.length < 1) {
45:             System.out.println("Usage: java WebReader url");
46:             System.exit(1);
47:         }
48:         try {
49:             WebReader app = new WebReader();
50:             app.getData(arguments[0]);
51:         } catch (MalformedURLException mue) {
52:             System.out.println("Bad URL: " + arguments[0]);
53:         }
54:     }
55: }
```

The WebReader application requires one command-line argument—a web address—that can be set in NetBeans in the project configuration (Run, Set Project Configuration, Customize).

You can choose any URL. Try http://www.timeapi.org/utc/now to read a simple text file that contains the current time, as shown in Figure 17.1.

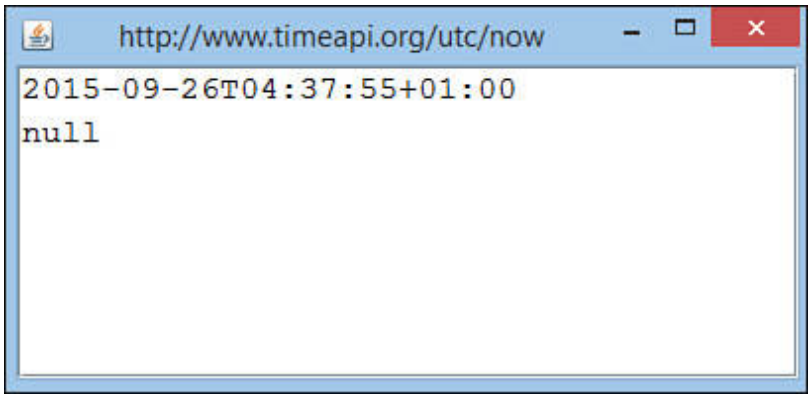


FIGURE 17.1 Running the WebReader application.

Two thirds of the `WebReader` class is devoted to running the application, creating the user interface, and creating a valid `URL` object. The web document is loaded over a stream and is displayed in a text area in the `getData()` method.

Four objects are used: `URL`, `HttpURLConnection`, `InputStreamReader`, and `BufferedReader`. These objects work together to pull the data from the Internet to the Java application. In addition, two objects are created to hold the data when it arrives: a `String` and a `StringBuilder`.

Lines 24–26 open an HTTP URL connection, which is necessary to get an input stream from that connection.

Lines 27–28 use the connection's `getContent()` method to create a new input stream reader. The method returns an input stream representing the connection to the URL.

Line 29 uses that input stream reader to create a new buffered input stream reader—a `BufferedReader` object called `buff`.

After you have this buffered reader, you can use its `readLine()` method to read a line of text from the input stream. The buffered reader puts characters in a buffer as they arrive and pulls them out of the buffer when requested.

The `do-while` loop in lines 32–36 reads the web document line by line, appending each line to the `StringBuilder` object created to hold the page's text.

After all the data has been read, line 37 converts the string builder into a string with the `toString()` method. Then it puts that result in the program's text area by calling the component's `setText(String)` method.

The `HttpUrlConnection` class includes several methods that affect the HTTP request or provide more information:

- `getHeaderField(int)` returns a string containing an HTTP header, such as `"Server"` (the web server hosting the document) or "Last-Modified" (the date the document was last changed). Headers are numbered from 0 upward. When the end of the headers is reached, this method returns `null`.
- `getHeaderFieldKey(int)` returns a string containing the name of the numbered header (such as "Server" or "Last-Modified") or `null`.
- `getResponseCode()` returns an integer containing the HTTP response code for the request, such as 200 (for valid requests) or 404 (for documents that could not be found).
- `getResponseMessage()` returns a string containing the HTTP response code and an explanatory message (such as "HTTP/1.0 200 OK"). The `HttpUrlConnection` class contains integer class variables for each of the valid response codes, including "HTTP_OK", "HTTP_NOT_FOUND", and "HTTP_MOVED_PERM".
- `getContentType()` returns a string containing the MIME type of the web document; some possible types are "text/html" for web pages and "text/xml" for XML files.
- `setFollowRedirects(boolean)` determines whether URL redirection requests should be followed (`true`) or ignored (`false`). When redirection is supported, a URL request can be forwarded by a web server from an obsolete URL to its correct address.

The following code could be added to WebReader's `getData()` method after line 26 to display headers along with the text of a document:

Click here to view code image

```
String key;
String header;
int i = 0;
do {
    key = conn.getHeaderFieldKey(i);
    header = conn.getHeaderField(i);
    if (key == null) {
```

```
            key = "";
        } else {
            key = key + ": ";
        }
        if (header != null) {
            text.append(key);
            text.append(header);
            text.append("\n");
        }
        i++;
    } while (header != null);
    text.append("\n");
```

Sockets

For networking applications beyond what the `URL` and `URLConnection` classes offer (for example, for other protocols or for more general networking applications), Java provides the `Socket` and `ServerSocket` classes as an abstraction of standard Transmission Control Protocol (TCP) socket programming techniques.

The `Socket` class provides a client-side socket interface similar to standard UNIX sockets. Create a new instance of `Socket` to open a connection, where *`hostName`* is the host to connect to and *`portNumber`* is the port number:

Click here to view code image

```
Socket connection = new Socket(hostName, portNumber);
```

After you create a socket, set its timeout value, which determines how long the application waits for data to arrive. This is handled by calling the socket's `setSoTimeOut(int)` method with the number of milliseconds to wait as the only argument:

Click here to view code image

```
connection.setSoTimeOut(50000);
```

When you use this method, any effort to read data from the socket represented by `connection` waits for only 50,000 milliseconds (50 seconds). If the timeout is reached, an `InterruptedIOException` is thrown, which gives you an opportunity in a `try-catch` block to either close the socket or try to read from it again.

If you don't set a timeout in a program that uses sockets, it might hang indefinitely, waiting for data.

Tip

This problem is usually avoided by putting network operations in their own thread and running them separately from the rest of the program, as covered during Day 7, "Exceptions and Threads."

After the socket is open, you can use input and output streams to read from and write to that socket:

Click here to view code image

```
BufferedInputStream bis = new
    BufferedInputStream(connection.getInputStream());
DataInputStream in = new DataInputStream(bis);

BufferedOutputStream bos = new
    BufferedOutputStream(connection.getOutputStream());
DataOutputStream out = new DataOutputStream(bos);
```

You don't need names for all these objects; they are used only to create a stream or stream reader. For an efficient shortcut, combine several statements, as in this example using a `Socket` object named `sock`:

Click here to view code image

```
DataInputStream in = new DataInputStream(
    new BufferedInputStream(
        sock.getInputStream()));
```

In this statement, the call to `sock.getInputStream()` returns an input stream associated with that socket. This stream is used to create a `BufferedInputStream`, and the buffered input stream is used to create a `DataInputStream`.

The only variables you are left with are `sock` and `in`, the two objects needed as you receive data from the connection and close it afterward. The intermediate objects—a `BufferedInputStream` and an `InputStream`—are needed only once.

After you're finished with a socket, don't forget to close it by calling the `close()` method. This also closes all the input and output streams you might have set up for that socket. For example:

```
connection.close();
```

Socket programming can be used for many services delivered using TCP/IP networking, including telnet, Simple Mail Transfer Protocol (SMTP) for incoming mail, WHOIS protocol for requesting domain name records, and Finger.

The last of these, Finger, is a protocol for asking a system about one of its users. By setting up a Finger server, a system administrator enables an Internet-connected machine to answer requests for user information. Users can provide information about themselves by creating `.plan` files, which are sent to anyone who uses Finger to find out more about them.

Although it has fallen into disuse because of security concerns, Finger was once a popular way for Internet users to share facts about themselves and their activities before blogs and social media. You could use Finger on a friend's account at another college or company to see whether that person was online and read the person's current `.plan` file.

As an exercise in socket programming, the Finger application is a rudimentary Finger client. Enter Listing 17.2 as a new class named `Finger` in NetBeans.

LISTING 17.2 The Full Text of `Finger.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.util.*;
 6:
 7: public class Finger {
 8:     public static void main(String[] args) {
 9:         String user;
10:         String host;
11:         if ((args.length == 1) && (args[0].indexOf("@") >
-1)) {
12:             StringTokenizer split = new
StringTokenizer(args[0],
13:                 "@");
14:             user = split.nextToken();
15:             host = split.nextToken();
16:         } else {
17:             System.out.println("Usage: java Finger
user@host");
18:             return;
19:         }
```

```
20:         try (Socket digit = new Socket(host, 79);
21:             BufferedReader in = new BufferedReader(
22:                 new
InputStreamReader(digit.getInputStream()));
23:             ) {
24:
25:             digit.setSoTimeout(20000);
26:             PrintStream out = new PrintStream(
27:                 digit.getOutputStream());
28:             out.print(user + "\015\012");
29:
30:             boolean eof = false;
31:             while (!eof) {
32:                 String line = in.readLine();
33:                 if (line != null) {
34:                     System.out.println(line);
35:                 } else {
36:                     eof = true;
37:                 }
38:             }
39:             digit.close();
40:         } catch (IOException e) {
41:             System.out.println("IO Error:" + e.getMessage());
42:         }
43:     }
44: }
```

When making a Finger request, specify a username followed by an at sign @ and a hostname, the same format as an email address. One of the last examples that still works is icculus@icculus.org, the Finger address of game developer Ryan Gordon. You can request his `.plan` file by running the Finger application with that address as the only command-line argument.

If icculus has an account on the icculus.org Finger server, running the Finger application displays his `.plan` file and perhaps other information. The server also lets you know when a user can't be found.

The output of this request is shown in Figure 17.2.

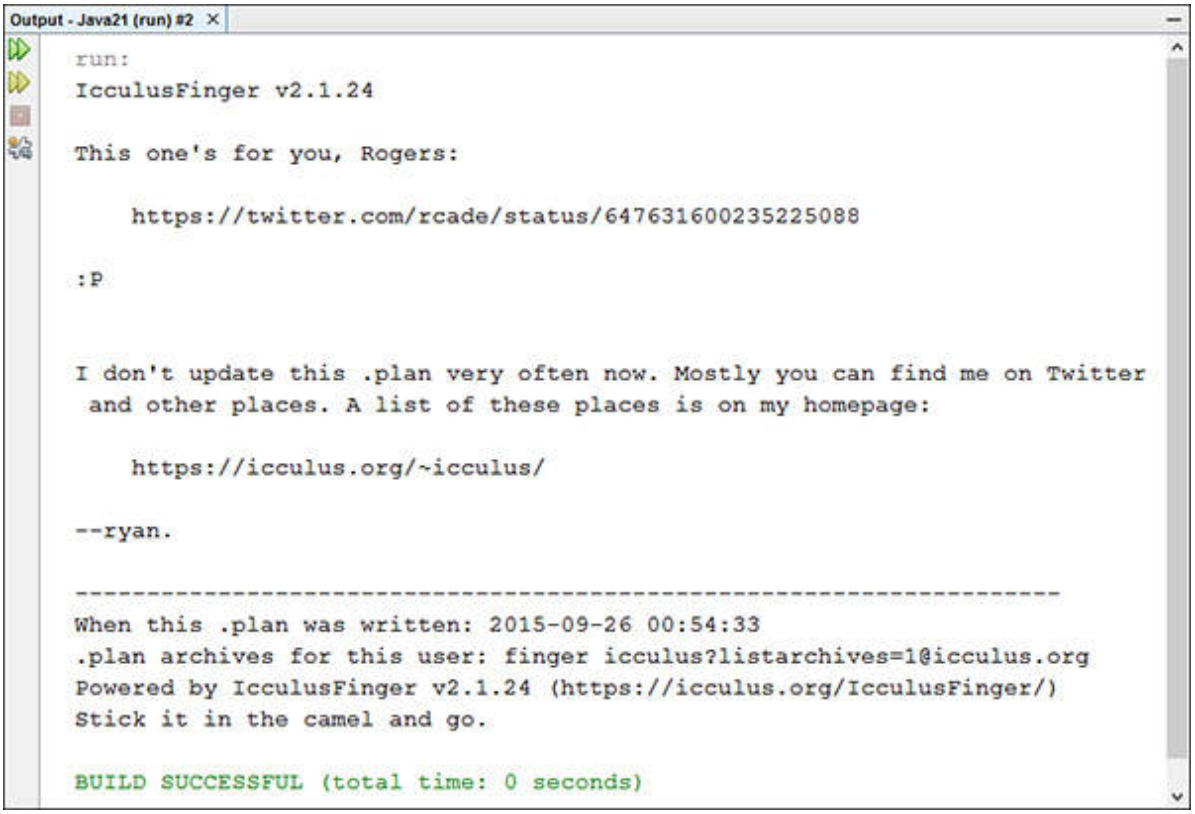


FIGURE 17.2 Making a Finger request using a socket.

The Finger application uses the `StringTokenizer` class to convert an address in *`user@host`* format into two `String` objects: `user` and `host` (lines 12–15).

The following socket activities are taking place:

- Line 20—A new `Socket` is created using the hostname and port 79, the port traditionally reserved for Finger services.
- Line 21–23—The socket is used to create an `InputStream`, which in turn is used to create a `BufferedReader`.
- Line 25—A timeout of 20 seconds is set for the socket.
- Line 26–27—The socket is used to get an `OutputStream`, which feeds into a new `PrintStream` object.
- Line 28—The Finger protocol requires that the username be sent through the socket, followed by a carriage return (`\015`) and linefeed (`\012`). This is handled by calling the `print()` method of the new print stream.
- Lines 31–38—The program loops as lines are read from the buffered reader. The end of output from the server causes `in.readLine()` to

return `null`, ending the loop.

The same techniques used to communicate with a Finger server through a socket can be used to connect to other popular Internet services. You could turn it into a telnet or web-reading client with a port change in Line 20 and little other modification.

Note

The Finger application makes use of the `try`-with-resources capability of Java in lines 20–23 of Listing 17.2. Declaring the socket and reader within the `try` statement's parentheses ensures that both of these resources will be closed even when the connection fails with an exception. This makes the explicit call to `close()` the socket in Line 39 unnecessary.

Socket Servers

Server-side sockets work similarly to client sockets, with the exception of the `accept()` method. A server socket listens on a TCP port for a connection from a client; when a client connects to that port, the `accept()` method accepts a connection from that client. By using both client and server sockets, you can create applications that communicate with each other over the network.

To create a server socket and bind it to a port, create a new instance of `ServerSocket` with a port number as an argument to the constructor, as in the following example:

Click here to view code image

```
ServerSocket servo = new ServerSocket(8888);
```

Use the `accept()` method to listen on that port (and to accept a connection from any clients if one is made):

```
servo.accept();
```

After the socket connection is made, you can use input and output streams to read from and write to the client.

To extend the behavior of the socket classes—for example, to allow network connections to work across a firewall or proxy—you can use the abstract class `SocketImpl` and the interface `SocketImplFactory` to create a

new transport-layer socket implementation. This approach allows those classes to be portable to other systems with different transport mechanisms. The problem with this mechanism is that although it works for simple cases, it prevents you from adding other protocols on top of TCP and from having multiple socket implementations for each Java runtime.

Because the `Socket` and `ServerSocket` classes are not final, you can create subclasses of these classes that use either the default socket implementation or your own implementation. This allows much more flexible network capabilities.

Designing a Server Application

Here's an example of a Java program that uses the `Socket` classes to implement a simple network-based server application.

The TimeServer application makes a connection to any client that connects to port 4415, displays the current time, and then closes the connection.

For an application to act as a server, it must monitor at least one port on the host machine for client connections. Port 4415 was chosen arbitrarily for this project, but it could be any number from 1024 to 65,535.

Note

The Internet Assigned Numbers Authority controls the usage of ports 0 to 1023, but claims are staked to the higher ports on a more informal basis. When choosing port numbers for your own client/server applications, it's a good idea to do research on what ports others are using. Search the Web for references to the port you want to use, and then search for the phrases "registered port numbers" and "well-known port numbers" to find lists of in-use ports. A good guide to port usage is available at www.sockets.com/services.htm.

When a client is detected, the server creates a `Date` object that represents the current date and time and then sends it to the client as a `String`.

In this exchange of information between the server and client, the server does almost all the work. The client's only responsibility is to establish a connection to the server and display messages received from the server.

Although you could develop a simple client for a project like this, you also can use any telnet application to act as the client, as long as it can connect to a

port you designate. (Windows includes a command-line application called telnet that you can use for this purpose.)

Listing 17.3 contains the full source code for the server application, a class called `TimeServer`.

LISTING 17.3 The Full Text of `TimeServer.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.util.*;
 6:
 7: public class TimeServer extends Thread {
 8:     private ServerSocket sock;
 9:
10:     public TimeServer() {
11:         super();
12:         try {
13:             sock = new ServerSocket(4415);
14:             System.out.println("TimeServer running ...");
15:         } catch (IOException e) {
16:             System.out.println("Error: couldn't create
socket.");
17:             System.exit(1);
18:         }
19:     }
20:
21:     public void run() {
22:         Socket client = null;
23:
24:         while (true) {
25:             if (sock == null)
26:                 return;
27:             try {
28:                 client = sock.accept();
29:                 BufferedOutputStream bb = new
BufferedOutputStream(
30:                     client.getOutputStream());
31:                 PrintWriter os = new PrintWriter(bb, false);
32:                 String outLine;
33:
34:                 Date now = new Date();
35:                 os.println(now);
36:                 os.flush();
37:
```

```
38:                 os.close();
39:                 client.close();
40:             } catch (IOException e) {
41:                 System.out.println("Error: couldn't
connect.");
42:                 System.exit(1);
43:             }
44:         }
45:     }
46:
47:     public static void main(String[] arguments) {
48:         TimeServer server = new TimeServer();
49:         server.start();
50:     }
51:
52: }
```

The TimeServer application creates a server socket on port 4415. When a client connects, a `PrintWriter` object is constructed from a buffered output stream so that a string—the current time—can be sent to the client.

After the string has been sent, the writer's `flush()` and `close()` methods end the data exchange and close the socket to await new connections.

Testing the Server

The TimeServer application must be running for a client to be able to connect to it. The server displays only one line of output if the application is running successfully, as shown in Figure 17.3.

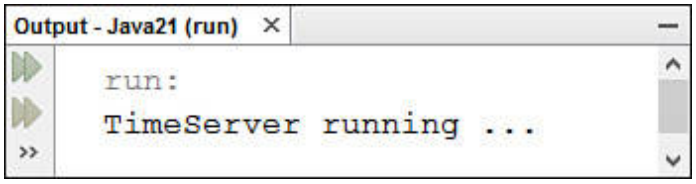


FIGURE 17.3 Launching an Internet server in a `ServerSocket`.

With the server running, you can connect to the server on port 4415 of your computer using a telnet program.

Do the following to run telnet on Windows:

- With Windows 8 and Windows 10, choose Start, choose the Search icon, and search for `telnet`. Click the `telnet` item to run it.
- With Windows 7 and Windows Vista, choose Start, All Programs, Accessories, Run to open the Run dialog box. Type `telnet` to run that

program. Then type the command `open localhost 4415` in the Open field and press Enter.

- With Windows XP and 2003, choose Start, Run to open the Run dialog box. Type `telnet` to run that program. Then type the command `open localhost 4415` in the Open field and press Enter.
- With earlier versions of Windows, choose Start, Run to open the Run dialog box, and then type `telnet` in the Open field and press Enter. A telnet window opens.

  To make a telnet connection using this program, select Connect, Remote System. A Connect dialog box opens. Enter `localhost` in the Host Name field, enter `4415` in the Port field, and leave the default value vt100 in the TermType field.

Caution

The telnet program may be disabled by default on Windows Vista and Windows 7. To enable it, open the Control Panel, choose Programs and Features, and click Turn Windows features on or off. The Windows Features dialog opens. Select the Telnet Client check box, and click OK.

The hostname `localhost` represents your own computer—the system running the application. You can use it to test server applications before deploying them permanently on the Internet.

Depending on how Internet connections have been configured on your system, you might need to log on to the Internet before a successful socket connection can be made between a telnet client and the TimeServer application.

If the server is on another computer connected to the Internet, you would specify that computer's hostname or IP address rather than `localhost`.

When you use telnet to make a connection with the TimeServer application, it displays the server's current time and closes the connection. The output of the telnet program should resemble Figure 17.4.

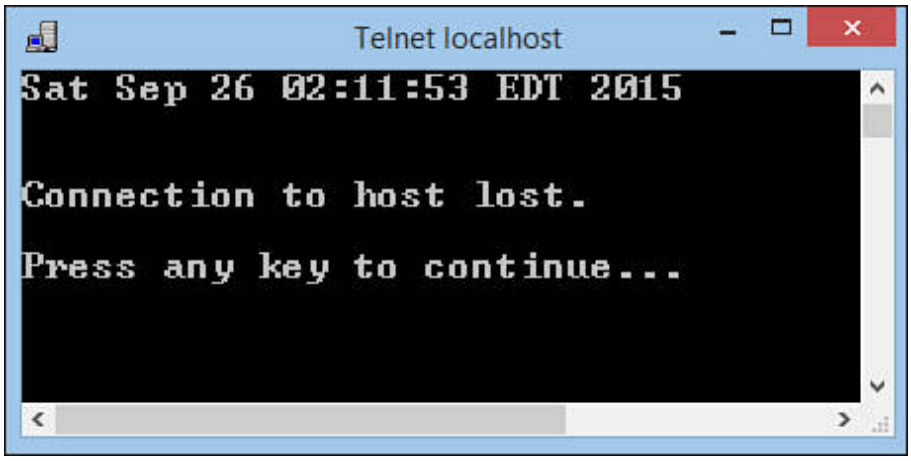


FIGURE 17.4 Making a telnet connection to your TimeServer.

The `java.nio` Package

The `java.nio` package expands the language's networking capabilities with classes useful for reading and writing data; working with files, sockets, and memory; and handling text.

Two related packages also are used often when you are working with the new input/output features: `java.nio.channels` and `java.nio.charset`.

Buffers

The `java.nio` package includes support for buffers—objects that represent data streams stored in memory.

Buffers often are used to improve the performance of programs that read input or write output. They enable a program to put a lot of data in memory, where it can be read, written, and modified more quickly.

A buffer corresponds with each of the primitive data types in Java:

- `ByteBuffer`
- `CharBuffer`
- `DoubleBuffer`
- `FloatBuffer`
- `IntBuffer`
- `LongBuffer`
- `ShortBuffer`

Each of these classes has a static method called `wrap()` that can be used to create a buffer from an array of the corresponding data type. The only argument to the method should be the array.

For example, the following statements create an array of integers and an `IntBuffer` that holds the integers in memory as a buffer:

Click here to view code image

```
int[] temperatures = { 90, 85, 87, 78, 80, 75, 70, 79, 85, 92 };
IntBuffer tempBuffer = IntBuffer.wrap(temperatures);
```

A buffer keeps track of how it is used, storing the position where the next item will be read or written. After the buffer is created, its `get()` method reads the data at the current position in the buffer. The following statements extend the previous example and display everything in the integer buffer:

Click here to view code image

```
for (int i = 0; tempBuffer.remaining() > 0; i++)
    System.out.println(tempBuffer.get());
```

Another way to create a buffer is to set up an empty buffer and then put data in it. To create the buffer, call the static method `allocate (int)` of the desired buffer class with the size of the buffer as an argument.

You can use five `put()` methods to store data in a buffer (or replace the data already there). The arguments used with these methods depend on the kind of buffer you're working with. These methods are used with an integer buffer:

- `put(int)` stores the integer at the current position in the buffer and then increments the position.
- `put(int, int)` stores an integer (the second argument) at a specific position in the buffer (the first argument).
- `put(int[])` stores all the elements of the integer array in the buffer, beginning at the first position in the buffer.
- `put(int[], int, int)` stores all or a portion of an integer array in the buffer. The second argument specifies the position in the buffer where the first integer in the array should be stored. The third argument specifies the number of elements from the array to store in the buffer.
- `put(IntBuffer)` stores the contents of an integer buffer in another buffer, beginning at the first position in the buffer.

As you put data in a buffer, you often must keep track of the current position so that you know where the next data will be stored.

To find out the current position, call the buffer's `position()` method. An integer is returned that represents the position. If this value is 0, you're at the start of the buffer.

Call the `position(int)` method to change the position to the argument specified as an integer.

Another important position to track when using buffers is the limit—the last place in the buffer that contains data.

It isn't necessary to figure out the limit when the buffer is always full; in that case, you know the buffer's last position has something in it.

However, if there's a chance your buffer might contain less data than you have allocated, you should call the buffer's `flip()` method after reading data into the buffer. This sets the current position to the start of the data you just read and sets the limit to the end.

If the buffer is 1,024 bytes in size and the page contains 1,500 bytes, the first attempt to read data loads the buffer with 1,024 bytes, filling it.

The second attempt to read data loads the buffer with only 476 bytes, leaving the rest empty. If you call `flip()` afterward, the current position is set to the beginning of the buffer, and the limit is set to 476.

The following code creates an array of Fahrenheit temperatures, converts them to Celsius, and then stores the Celsius values in a buffer:

Click here to view code image

```
int[] temps = { 90, 85, 87, 78, 80, 75, 70, 79, 85, 92, 99 };
IntBuffer tempBuffer = IntBuffer.allocate(temps.length);
for (int i = 0; i < temps.length; i++) {
    float celsius = ( (float) temps[i] - 32 ) / 9 * 5;
    tempBuffer.put( (int) celsius );
}
tempBuffer.position(0);
for (int i = 0; tempBuffer.remaining() > 0; i++) {
    System.out.println(tempBuffer.get());
}
```

After the buffer's position is set back to the start, the buffer's contents are displayed.

Byte Buffers

You can use the buffer methods introduced so far with byte buffers, but byte buffers also offer additional useful methods.

For starters, byte buffers have methods to store and retrieve data that isn't a byte:

- `putChar(`*`char`*`)` stores 2 bytes in the buffer that represent the specified `char` value.
- `putDouble(`*`double`*`)` stores 8 bytes in the buffer that represent a `double` value.
- `putFloat(`*`float`*`)` stores 4 bytes in the buffer that represent a `float` value.
- `putInt(`*`int`*`)` stores 4 bytes in the buffer that represent an `int` value.
- `putLong(`*`long`*`)` stores 8 bytes in the buffer that represent a `long` value.
- `putShort(`*`short`*`)` stores 2 bytes in the buffer that represent a `short` value.

Each of these methods puts more than 1 byte in the buffer, moving the current position forward by the same number of bytes.

There also are methods to retrieve nonbytes from a byte buffer: `getChar()`, `getDouble()`, `getFloat()`, `getInt()`, `getLong()`, and `getShort()`.

Character Sets

Character sets, which are offered in the `java.nio.charset` package, are a set of classes used to convert data between byte buffers and character buffers.

The three main classes are as follows:

- `Charset` is a Unicode character set with a different byte value for each different character in the set.
- `CharsetDecoder` is a class that transforms a series of bytes into a series of characters.
- `CharsetEncoder` is a class that transforms a series of characters into a series of bytes.

Before you can perform any transformations between byte and character buffers, you must create a `Charset` object that maps characters to their corresponding byte values.

To create a character set, call the `forName(`*`String`*`)` static method of the `Charset` class, specifying the name of the set's character encoding.

Java supports six character encodings:

- **US-ASCII**—The 128-character ASCII set that makes up the Basic Latin block of Unicode (also called ISO646-US)
- **ISO-8859-1**—The 256-character ISO Latin Alphabet No. 1 character set (also called ISO-LATIN-1)
- **UTF-8**—A character set that includes US-ASCII and the Universal Character Set (also called Unicode), a set composed of thousands of characters used in the world's languages
- **UTF-16BE**—The Universal Character Set represented as 16-bit characters with bytes stored in big-endian byte order
- **UTF-16LE**—The Universal Character Set represented as 16-bit characters with bytes stored in little-endian byte order
- **UTF-16**—The Universal Character Set represented as 16-bit characters with the order of bytes indicated by an optional byte-order mark

The following statement creates a `Charset` object for the ISO-8859-1 character set:

Click here to view code image

```
Charset isoset = Charset.forName("ISO-8859-1");
```

After you have a character set object, you can use it to create encoders and decoders. Call the object's `newDecoder()` method to create a `CharsetDecoder` and the `newEncoder()` method to create a `CharsetEncoder`.

To transform a byte buffer into a character buffer, call the decoder's `decode(`*`ByteBuffer`*`)` method, which returns a `CharBuffer` containing the bytes transformed into characters.

To transform a character buffer into a byte buffer, call the encoder's `encode(`*`CharBuffer`*`)` method. A `ByteBuffer` is returned containing the characters' byte values.

The following statements convert a byte buffer called `netBuffer` into a character buffer using the ISO-8859-1 character set:

Click here to view code image

```
ByteBuffer netBuffer = ByteBuffer.allocate(20480);
// code to fill byte buffer would be here
Charset set = Charset.forName("ISO-8859-1");
CharsetDecoder decoder = set.newDecoder();
netBuffer.position(0);
CharBuffer netText = decoder.decode(netBuffer);
```

Caution

Before the decoder is used to create the character buffer, the call to `position(0)` resets the current position of the `netBuffer` to the start. When you're working with buffers for the first time, it's easy to overlook this, resulting in a buffer with much less data than you expected.

Channels

A common use for a buffer is to associate it with an input or output stream. You can fill a buffer with data from an input stream or write a buffer to an output stream.

To do this, you must use a channel—an object that connects a buffer to the stream. Channels are part of the `java.nio.channels` package.

You can associate channels with a stream by calling the `getChannel()` method available in some of the stream classes in the `java.io` package.

The `FileInputStream` and `FileOutputStream` classes have `getChannel()` methods that return a `FileChannel` object. This file channel can be used to read, write, and modify the data in the file.

The following statements create a file input stream and a channel associated with that file:

Click here to view code image

```
try {
    String source = "prices.dat";
    FileInputStream inSource = new FileInputStream(source);
    FileChannel inChannel = inSource.getChannel();
} catch (FileNotFoundException fne) {
```

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

After you have created the file channel, you can find out how many bytes the file contains by calling its `size()` method. This is necessary if you want to create a byte buffer to hold the file's contents.

Bytes are read from a channel into a `ByteBuffer` with the `read(ByteBuffer, long)` method. The first argument is the buffer. The second argument is the current position in the buffer, which determines where the file's contents will begin to be stored.

The following statements extend the last example by reading a file into a byte buffer using the `inChannel` file channel:

Click here to view code image

```
long inSize = inChannel.size();
ByteBuffer data = ByteBuffer.allocate( (int) inSize );
inChannel.read(data, 0);
data.position(0);
for (int i = 0; data.remaining() > 0; i++) {
    System.out.print(data.get() + " ");
}
```

The attempt to read from the channel generates an `IOException` error if a problem occurs. Although the byte buffer is the same size as the file, this isn't a requirement. If you are reading the file into the buffer so that you can modify it, you can allocate a larger buffer.

The next project you undertake incorporates the new input/output features you have learned about so far: buffers, character sets, and channels.

The BufferConverter application reads a small file into a byte buffer, displays the contents of the buffer, converts it to a character buffer, and then displays the characters.

Enter the code shown in Listing 17.4 as the new Java class `BufferConverter` in the `com.java21days` package.

LISTING 17.4 The Full Text of `BufferConverter.java`

Click here to view code image

```
1: package com.java21days;
2:
3: import java.nio.*;
```

```
 4: import java.nio.channels.*;
 5: import java.nio.charset.*;
 6: import java.io.*;
 7:
 8: public class BufferConverter {
 9:     public static void main(String[] arguments) {
10:         try {
11:             // read byte data into a byte buffer
12:             String data = "friends.dat";
13:             FileInputStream inData = new
FileInputStream(data);
14:             FileChannel inChannel = inData.getChannel();
15:             long inSize = inChannel.size();
16:             ByteBuffer source = ByteBuffer.allocate((int)
inSize);
17:             inChannel.read(source, 0);
18:             source.position(0);
19:             System.out.println("Original byte data:");
20:             for (int i = 0; source.remaining() > 0; i++) {
21:                 System.out.print(source.get() + " ");
22:             }
23:             // convert byte data into character data
24:             source.position(0);
25:             Charset ascii = Charset.forName("US-ASCII");
26:             CharsetDecoder toAscii = ascii.newDecoder();
27:             CharBuffer destination = toAscii.decode(source);
28:             destination.position(0);
29:             System.out.println("\n\nNew character data:");
30:             for (int i = 0; destination.remaining() > 0; i++)
{
31:                 System.out.print(destination.get());
32:             }
33:             System.out.println();
34:         } catch (FileNotFoundException fne) {
35:             System.out.println(fne.getMessage());
36:         } catch (IOException ioe) {
37:             System.out.println(ioe.getMessage());
38:         }
39:     }
40: }
```

Before you run the file, you need a copy of `friends.dat`, the small file of byte data used in the application. To download it from the book's website at www.java21days.com, open the Day 17 page, right-click the `friends.dat` hyperlink, and save the file in a folder on your computer.

To copy that file into the same place as the application, follow these steps in NetBeans:

- In the folder where you downloaded `friends.dat`, right-click the file and choose Copy.
- In NetBeans, click the Files pane to bring it to the front.
- Right-click Java21 (the top folder in the pane) and choose Paste.

The file will be stored in the project's main folder.

Tip

You also can create your own file. In NetBeans, choose File, New File. In the New File dialog, choose the category Other and the file type Empty File. Give it the filename `friends.dat`. In the source code editor, type a sentence or two in the document, and save the file.

If you use the copy of `friends.dat` from the book's website, the output of the BufferConverter application is shown in Figure 17.5.

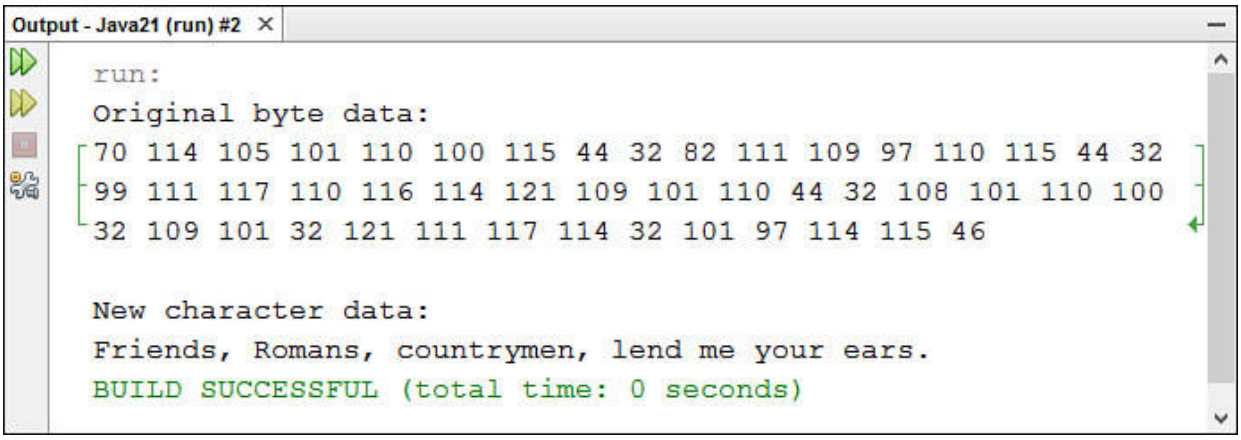


FIGURE 17.5 Reading character data from a buffer.

The BufferConverter application uses the techniques introduced today to read data and represent it as bytes and characters, but you could have accomplished the same thing with the original input/output package, `java.io`.

For this reason, you might wonder why it's worth learning the new package at all.

One reason is that buffers enable you to manipulate large amounts of data much more quickly. You'll find out another reason in the next section.

Network Channels

A popular feature of the `java.nio` package is its support for nonblocking input and output over a networking connection.

In Java, blocking refers to a statement that must complete execution before anything else happens in the program. All the socket programming you have done up to this point has used blocking methods exclusively. For example, in the TimeServer application, when the server socket's `accept()` method is called, nothing else happens in the program until a client makes a connection.

As you can imagine, it's problematic for a networking program to wait until a particular statement is executed, because numerous things can go wrong. Connections can be broken. A server could go offline. A socket connection could appear to be stalled because a blocked statement is waiting for something to happen.

For example, a client application that reads and buffers data over HTTP might be waiting for a buffer to be filled even though no more data remains to be sent. The program will appear to have halted, because the blocked statement never finishes executing.

With the `java.nio` package, you can create networking connections and read to and write from them using nonblocking methods.

Here's how it works:

- Associate a socket channel with an input or output stream.
- Configure the channel to recognize the kind of networking events you want to monitor, such as new connections, attempts to read data over the channel, and attempts to write data.
- Call a method to open the channel.
- Because the method is nonblocking, the program continues executing so that you can handle other tasks.
- If one of the networking events you are monitoring takes place, your program is notified—a method associated with the event is called.

This is comparable to how user-interface components are programmed in Swing. An interface component is associated with one or more event listeners and is placed in a container. If the interface component receives input being monitored by a listener, an event-handling method is called. Until that happens, the program can handle other tasks.

To use nonblocking input and output, you must work with channels instead of streams.

Nonblocking Socket Clients and Servers

The first step in developing a nonblocking client or server is creating an object that represents the Internet address to which you are connecting. This task is handled by the `InetSocketAddress` class in the `java.net` package.

If the server is identified by a hostname, call `InetSocketAddress(`*`String`*`, `*`int`*`)` with two arguments: the server's name and port number.

If the server is identified by its IP address, use the `InetAddress` class in `java.net` to identify the host. Call the static method `InetAddress.getByName(`*`String`*`)` with the host's IP address as the argument. The method returns an `InetAddress` object representing the address, which you can use in calling `InetSocketAddress(`*`InetAddress`*`, `*`int`*`)`. The second argument is the server's port number.

Nonblocking connections require a socket channel, another of the classes in the `java.nio` package. Call the `open()` static method of the `SocketChannel` class to create the channel.

A socket channel can be configured for blocking or nonblocking communication. To set up a nonblocking channel, call the channel's `configureBlocking (`*`boolean`*`)` method with an argument of `false`. Calling it with `true` makes it a blocking channel.

After the channel is configured, call its `connect(`*`InetSocketAddress`*`)` method to connect the socket.

On a blocking channel, the `connect()` method attempts to establish a connection to the server and waits until it is complete, returning a value of `true` to indicate success.

On a nonblocking channel, the `connect()` method returns immediately with a value of `false`. To figure out what's going on over the channel and respond to events, you must use a channel-listening object called a `Selector`.

A `Selector` is an object that keeps track of things that happen to a socket channel (or another channel in the package that is a subclass of `SelectableChannel`).

To create a `Selector`, call its `open()` method, as in the following statement:

Click here to view code image

```
Selector monitor = Selector.open();
```

When you use a `Selector`, you must indicate the events you want to monitor. You do so by calling a channel's `register(`*`Selector`*`, `*`int`*`, `*`Object`*`)` method.

The three arguments to `register()` are the following:

- The `Selector` object you have created to monitor the channel
- An `int` value that represents the events being monitored (also called selection keys)
- An `Object` that can be delivered along with the key, or `null` otherwise

Instead of using an integer value as the second argument, it's easier to use one or more class variables from the `SelectionKey` class: `SelectionKey.OP_CONNECT` to monitor connections, `SelectionKey.OP_READ` to monitor attempts to read data, and `SelectionKey.OP_WRITE` to monitor attempts to write data.

The following statements create a `Selector` to monitor a socket channel called `wire` for reading data:

Click here to view code image

```
Selector spy = Selector.open();
channel.register(spy, SelectionKey.OP_READ, null);
```

To monitor more than one kind of key, add together the `SelectionKey` class variables. For example:

Click here to view code image

```
Selector spy = Selector.open();
channel.register(spy, SelectionKey.OP_READ +
SelectionKey.OP_WRITE,
    null);
```

After the channel and selector have been set up, you can wait for events by calling the selector's `select()` or `select(long)` methods.

The `select()` method is a blocking method that waits until something has happened on the channel.

The `select(long)` method is a blocking method that waits until something has happened or the specified number of milliseconds has passed, whichever comes first.

Both `select()` methods return the number of events that have taken place, or `0` if nothing has happened. You can use a `while` loop with a call to the `select()` method as a way to loop until something happens on the channel.

After an event has taken place, you can find out more about it by calling the selector's `selectedKeys()` method, which returns a `Set` object containing details on each of the events.

Use this `Set` object as you would any other set, creating an `Iterator` to move through the set by using its `hasNext()` and `next()` methods.

The call to the set's `next()` method returns an object that should be cast to a `SelectionKey`. This object represents an event that took place on the channel.

Three methods in the `SelectionKey` class can be used to identify the key in a client program: `isReadable ()`, `isWritable ()`, and `isConnectible ()`. Each returns a `boolean` value. (A fourth method is used when you're writing a server: `isAcceptable ()`.)

After you retrieve a key from the set, call the key's `remove()` method to indicate that you will do something with it.

The last thing to find out about the event is the channel on which it took place. Call the key's `channel()` method, which returns the associated `SocketChannel`.

If one of the events identifies a connection, you must make sure that the connection has been completed before using the channel. Call the key's `isConnectionPending()` method, which returns `true` if the connection is still in progress and `false` if it is complete.

To deal with a connection that is still in progress, you can call the socket's `finishConnect()` method, which attempts to complete the connection.

Using a nonblocking socket channel involves the interaction of numerous new classes from the `java.nio` and `java.net` packages.

To give you a more complete picture of how these classes work together, the day's final project is FingerServer, a web application that uses a nonblocking socket channel to handle Finger requests.

Enter the code shown in Listing 17.5 as the class `FingerServer` in the package `com.java21days` and save the application.

LISTING 17.5 The Full Text of `FingerServer.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.nio.channels.*;
 6: import java.util.*;
 7:
 8: public class FingerServer {
 9:
10:     public FingerServer() {
11:         try {
12:             // Create a non-blocking server socket channel
13:             ServerSocketChannel sock =
ServerSocketChannel.open();
14:             sock.configureBlocking(false);
15:
16:             // Set the host and port to monitor
17:             InetSocketAddress server = new
InetSocketAddress(
18:                 "localhost", 79);
19:             ServerSocket socket = sock.socket();
20:             socket.bind(server);
21:
22:             // Create the selector and register it on the
channel
23:             Selector selector = Selector.open();
24:             sock.register(selector, SelectionKey.OP_ACCEPT);
25:
26:             // Loop forever, looking for client connections
27:             while (true) {
28:                 // Wait for a connection
29:                 selector.select();
30:
31:                 // Get list of selection keys with pending
events
32:                 Set keys = selector.selectedKeys();
33:                 Iterator it = keys.iterator();
```

```
 34:
 35:                 // Handle each key
 36:                 while (it.hasNext()) {
 37:
 38:                     // Get the key and remove it from the
iteration
 39:                     SelectionKey sKey = (SelectionKey)
it.next();
 40:
 41:                     it.remove();
 42:                     if (sKey.isAcceptable()) {
 43:
 44:                         // Create a socket connection with
client
 45:                         ServerSocketChannel selChannel =
 46:                             (ServerSocketChannel)
sKey.channel();
 47:                         ServerSocket sSock =
selChannel.socket();
 48:                         Socket connection = sSock.accept();
 49:
 50:                         // Handle the Finger request
 51:                         handleRequest(connection);
 52:                         connection.close();
 53:                     }
 54:                 }
 55:             }
 56:          } catch (IOException ioe) {
 57:              System.out.println(ioe.getMessage());
 58:          }
 59:       }
 60:
 61:     private void handleRequest(Socket connection)
 62:         throws IOException {
 63:
 64:         // Set up input and output
 65:         InputStreamReader isr = new InputStreamReader (
 66:             connection.getInputStream());
 67:         BufferedReader is = new BufferedReader(isr);
 68:         PrintWriter pw = new PrintWriter(new
 69:             BufferedOutputStream(connection.getOutputStream(
)),
 70:             false);
 71:
 72:         // Output server greeting
 73:         pw.println("Nio Finger Server");
 74:         pw.flush();
 74:
 75:         // Handle user input
 76:         String outLine = null;
 77:         String inLine = is.readLine();
```

```
 78:
 79:         if (inLine.length() > 0) {
 80:             outLine = inLine;
 81:         }
 82:         readPlan(outLine, pw);
 83:
 84:         // Clean up
 85:         pw.flush();
 86:         pw.close();
 87:         is.close();
 88:     }
 89:
 90:     private void readPlan(String userName, PrintWriter pw) {
 91:         try {
 92:             FileReader file = new FileReader(userName +
".plan");
 93:             BufferedReader buff = new BufferedReader(file);
 94:             boolean eof = false;
 95:
 96:             pw.println("\nUser name: " + userName + "\n");
 97:
 98:             while (!eof) {
 99:                 String line = buff.readLine();
100:
101:                 if (line == null) {
102:                     eof = true;
103:                 } else {
104:                    pw.println(line);
105:                 }
106:             }
107:
108:             buff.close();
109:         } catch (IOException e) {
110:             pw.println("User " + userName + " not found.");
111:         }
112:     }
113:
114:     public static void main(String[] arguments) {
115:         FingerServer nio = new FingerServer();
116:     }
117: }
```

The Finger server requires one or more user `.plan` files stored in text files. These files should have names that take the form *`username`*`.plan`—for example, `linus.plan`, `lucy.plan`, and `franklin.plan`. Before running the server, create one or more plan files in the root folder of the Java21 project.

When you're done, run the Finger server. The application waits for incoming Finger requests, creating a nonblocking server socket channel and registering one kind of key for a selector to look for: connection events.

Inside a `while` loop that begins on Line 27, the server calls the `Selector` object's `select()` method to see whether the selector has received any keys, which would occur when a Finger client makes a connection. When it has, `select()` returns the number of keys, and the statements inside the loop are executed.

After the connection is made, a buffered reader is created to hold a request for a `.plan` file. The syntax for the command is the username of the `.plan` file being requested.

While the Finger server is running, you can test this application with the Finger client. Create a custom project configuration in NetBeans to set the command-line argument of the Finger user:

- Choose Run, Set Project Configuration, Customize. The Project Properties dialog opens.
- In the Main Class text field, enter `Finger`.
- In the Arguments text field, enter `franklin@localhost`, and click OK.
- Run the application by choosing Run, Run Project.

The output is shown in Figure 17.6 when you request the user franklin on the computer localhost.

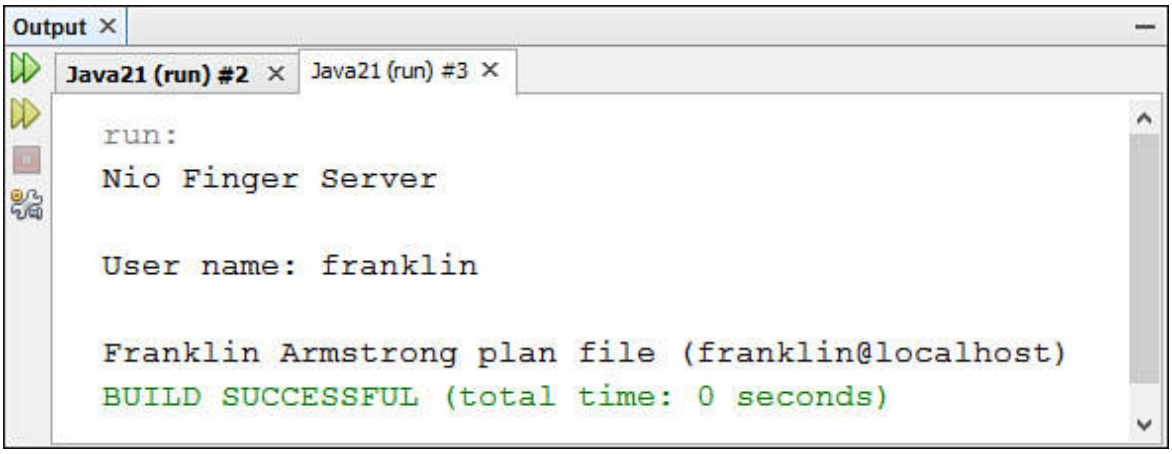


FIGURE 17.6 Making a Finger request from your Finger server.

Run the application again with `lucy@localhost` to see Lucy's `.plan` file, and finally with `linus@localhost` to look for Linus.

When you're done with the Finger server, press the Stop button on the left edge of the Output pane to shut it down.

Caution

The plan files must be in the root folder of the Java21 project for the FingerServer application to find them. If they were saved somewhere else, you can move them by dragging and dropping in NetBeans. Click the Files tab in the Projects pane to see a list of the project's files. Find the plan files, and drag them to the same folder that holds `friends.dat`.

Summary

Today you learned how to use URLs, URL connections, and input streams in combination to pull data from the Web into your program.

Networking can be extremely useful. The WebReader project is a rudimentary web browser. It can load a web page or RSS file into a Java program and display it. However, it doesn't do anything to make sense of the markup tags, presenting the raw text delivered by a web server.

You created a socket application that implements the basics of the Finger protocol, a method for retrieving user information on the Internet.

You also learned how client and server programs are written in Java using the nonblocking techniques in the `java.nio` package.

To use nonblocking techniques, you learned about the fundamental classes of Java's new networking package: buffers, character encoders and decoders, socket channels, and selectors.

Q&A

Q Can other computers connect to my Finger server over the Internet?

A Probably not. Most computers have firewall settings and router security settings that would not accept incoming requests on port 79, the one used by the Finger protocol.

If you create a server that isn't just for testing purposes, you must figure out how to configure the firewall and router to allow the server to

access all the ports that it requires.

Because Internet servers are frequent targets of attack, you must make sure your server can handle malformed client requests and other hacking attempts. You also should run the server with a user account that only has access to the files and system resources it needs and no others. This prevents a hacker from compromising the server and using it to read confidential data, infect the computer with viruses, and launch other harmful exploits.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which of the following is *not* an advantage of the new `java.nio` package and its related packages?

   A. Large amounts of data can be manipulated quickly with buffers.

   B. Networking connections can be nonblocking for more reliable use in your applications.

   C. Streams are no longer necessary to read and write data over a network.

2. In the Finger protocol, which program makes a request for information about a user?

   A. The client

   B. The server

   C. Both can make that request.

3. Which method is preferred for loading the data from a web page into your Java application?

   A. Creating a `Socket` and an input stream from that socket

   B. Creating a `URL` and an `HttpURLConnection` from that object

   C. Loading the page using the method `toString()`

Answers

1. C. The `java.nio` classes work in conjunction with streams. They don't replace them.
2. A. The client requests information, and the server sends back something in response. This is traditionally how client/server applications function, although some programs can act as both client and server.
3. B. Sockets are good for low-level connections, such as when you are implementing a new protocol. For existing protocols such as HTTP, some classes are better suited to that protocol—`URL` and `HttpURLConnection`, in this case.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
import java.nio.*;

public class ReadTemps {
    public ReadTemps() {
        int[] temperatures = { 78, 80, 75, 70, 79, 85, 92, 99, 90
};
        IntBuffer tempBuffer = IntBuffer.wrap(temperatures);
        int[] moreTemperatures = { 65, 44, 71 };
        tempBuffer.put(moreTemperatures);
        System.out.println("First int: " + tempBuffer.get());
    }
}
```

What will be the output when this application is run?

A. `First int: 78`

B. `First int: 71`

C. `First int: 70`

D. None of the above

The answer is available on the book's website at www.java21days.com. Visit the Day 17 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Write an application that stores some of your favorite web pages on your computer so that you can read them while you are not connected to the Internet.
2. Modify the FingerServer application to use the `try`-with-resources improvement to `try-catch` blocks in Java.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 18. Accessing Databases with JDBC 4.2 and Derby

Almost all Java programs deal with data in some way. So far you have used primitive types, objects, arrays, hash maps, and other data structures.

Today, you work with data in a more sophisticated way by exploring Java Database Connectivity (JDBC), a class library that connects Java programs to relational databases.

Java includes Java DB, a compact relational database that makes it easier than ever to incorporate a database into your applications. Java DB is Oracle's name for Apache Derby, an open source database maintained by the Apache Software Foundation.

Today, you explore JDBC in the following ways:

- Using JDBC drivers to work with different relational databases
- Accessing a database with Structured Query Language (SQL)
- Reading records from a database using SQL and JDBC
- Adding records to a database using SQL and JDBC
- Creating a new Java DB database and reading its records

Java Database Connectivity

Java Database Connectivity (JDBC) is a set of classes that can be used to develop client/server applications that work with databases developed by Microsoft, Sybase, Oracle, IBM, and other sources.

With JDBC, you can use the same methods and classes in Java programs to read and write records and perform other kinds of database access. A class called a *driver* acts as a bridge to the database source. There are drivers for each of the popular databases.

Client/server software connects a user of information with a provider of that information, and it's one of the most common forms of programming. You use it every time you use the Web: A web browser client requests pages, image files, and other documents using a uniform resource locator (URL). Web servers provide the requested information, if it can be found, for the client.

One of the biggest obstacles faced by database programmers is the wide variety of database formats in use, each with its own proprietary method of accessing data.

To simplify using relational database programs, a standard language called Structured Query Language (SQL) was developed. This language supplants the need to learn different database-querying languages for each database format. Java DB supports SQL.

In database programming, a request for records in a database is called a *query*. Using SQL, you can send complex queries to a database and get the records you're looking for in any order you specify.

Consider the example of a database programmer at a student loan company who has been asked to prepare a report on the most delinquent loan recipients. The programmer could use SQL to query a database for all records in which the last payment was more than 180 days ago and the amount due is more than $0.00. SQL also can be used to control the order in which records are returned, so the programmer can get the records in the order of Social Security number, recipient name, amount owed, or another field in the loan database.

All this is possible with SQL. The programmer doesn't need any of the proprietary languages associated with popular database formats.

Caution

SQL is supported by many database tools, so, in theory, you should be able to use the same SQL commands for each database tool that supports the language. However, you will still need to learn the idiosyncrasies of a specific database tool when accessing it through SQL.

SQL is the industry-standard approach to accessing relational databases. JDBC supports SQL, enabling developers to use a wide range of database formats without knowing the specifics of the underlying database. JDBC also supports the use of database queries specific to a database format.

The JDBC class library's approach to accessing databases with SQL is comparable to existing database-development techniques, so interacting with a SQL database by using JDBC isn't much different from using traditional

database tools. Java programmers who already have some database experience can hit the ground running with JDBC.

The JDBC library includes classes for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating a statement using SQL
- Executing that SQL query in the database
- Viewing the resulting records

These JDBC classes all are part of the `java.sql` package.

Database Drivers

Java programs that use JDBC classes can follow the familiar programming model of issuing SQL statements and processing the resulting data. The format of the database and the platform it was prepared on don't matter.

This platform and database independence is made possible by a driver manager. The classes of the JDBC library are largely dependent on driver managers, which keep track of the drivers required to access database records. You need a different driver for each database format that's used in a program, and sometimes you might need several drivers for versions of the same format. Java DB includes its own driver.

JDBC also includes a driver that bridges JDBC and another database-connectivity standard, ODBC.

Examining a Database

NetBeans has extensive support for database programming. Before you begin writing code, you can use it to connect to a database, learn about the tables it contains, and see the data in those tables.

To connect to a Java DB database, first you must start the database server.

In the Projects pane, click the Services tab to bring it to the front, as shown in Figure 18.1. The Databases item includes a Java DB item. Right-click it and choose Start Server.

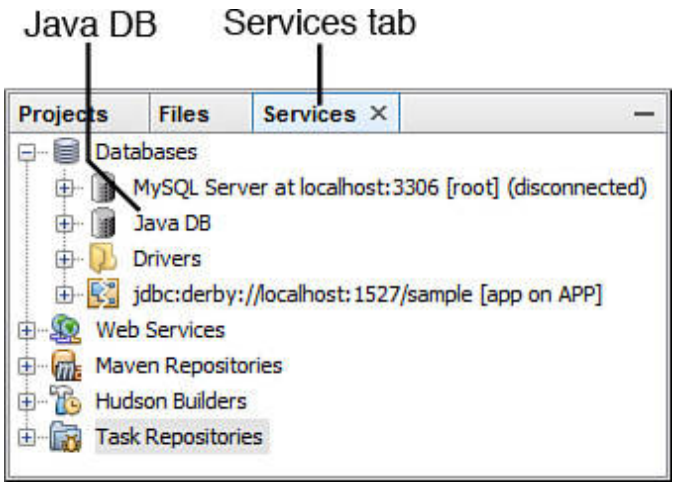


FIGURE 18.1 Starting the Java DB database server.

The first time you start a Java DB server in NetBeans, it might fail with a security error. In that circumstance, NetBeans displays a balloon dialog reporting a Security Manager Problem (Figure 18.2).

FIGURE 18.2 Dealing with a Security Manager error.

There are no significant security risks when running Java DB to develop and test JDBC applications in this chapter. Click the Disable Security Manager button; then start Java DB again in the Services tab of the Projects pane. Right-click Java DB and choose Start Server.

When Java DB launches after any security issues are resolved, it launches and displays a few lines of text to indicate what it's doing. This output is shown in Figure 18.3.

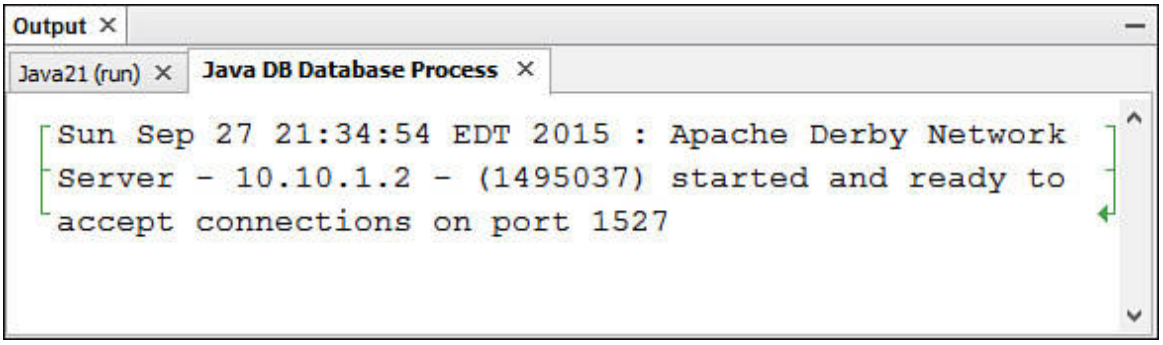


FIGURE 18.3 Launching a Java DB server with NetBeans.

This database server calls itself the Apache Derby Network Server, a reflection of the fact that Oracle's Java DB is an implementation of Derby.

The server's output indicates that the server is running on port 1527 and is ready to take connections. Keep this window open so that you can monitor the server while it runs.

In the Services pane under Java DB is a sample database named `sample`. Connect to this database by right-clicking `sample` and choosing Connect.

An item in the Services pane changes from a broken icon into an unbroken one: jdbc:derby://localhost:1527/sample.

This is an active connection to the database. Expand this item, and then expand `APP`, `Tables`, and `CUSTOMER`. A list of fields in the `CUSTOMER` table appears, as shown in Figure 18.4.

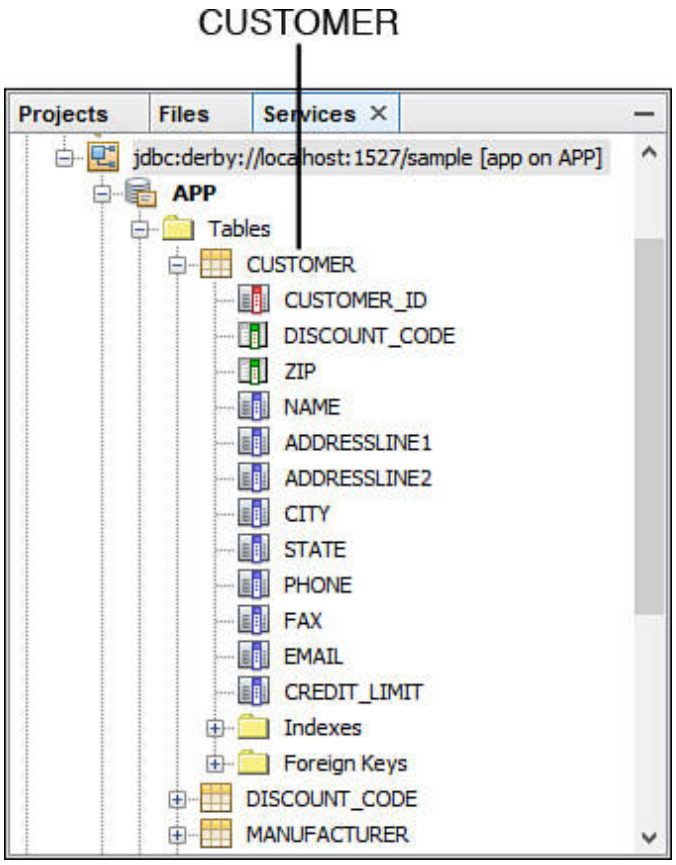


FIGURE 18.4 Examining tables in a Java DB database.

You can view the records in this table by right-clicking `CUSTOMER` and choosing View Data. Two things appear in other panes on NetBeans.

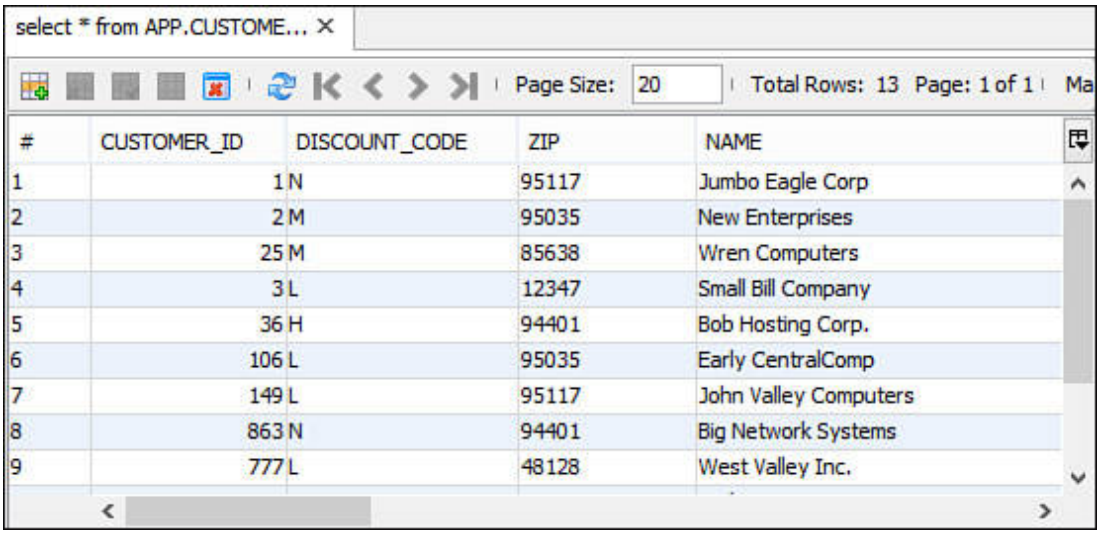
A SQL command appears where the source code editor normally appears:

```
select * from APP.CUSTOMER
```

This command, which is called a SQL query, selects all fields from `APP.CUSTOMER`. The asterisk character * could be replaced with the name of one or more fields, separated by commas.

Another pane displays the result of this command: all the data in this table, organized into rows and columns. Each column is a field, and each row is a record in the table.

Figure 18.5 shows the contents of the `CUSTOMER` table. This is the table you'll be writing Java code to access.



#	CUSTOMER_ID	DISCOUNT_CODE	ZIP	NAME
1	1	N	95117	Jumbo Eagle Corp
2	2	M	95035	New Enterprises
3	25	M	85638	Wren Computers
4	3	L	12347	Small Bill Company
5	36	H	94401	Bob Hosting Corp.
6	106	L	95035	Early CentralComp
7	149	L	95117	John Valley Computers
8	863	N	94401	Big Network Systems
9	777	L	48128	West Valley Inc.

FIGURE 18.5 Displaying database records in a table.

Reading Records from a Database

Your first project today is a Java application that connects to a sample Java DB database included with NetBeans and that reads records from a table.

Working with a database in a Java program is relatively easy if you are conversant with SQL.

The first task in a JDBC program is to load the driver (or drivers) that will be used to connect to a data source. A driver is loaded with the `Class.forName(String)` method. `Class`, part of the `java.lang` package, can be used to load classes into the Java Virtual Machine (JVM). The `forName(String)` method loads the class named by the specified string. This method can throw a `ClassNotFoundException`.

Programs that use Java DB can use `org.apache.derby.jdbc.ClientDriver`, a driver included with the database. Loading this class into the JVM requires the following statement:

Click here to view code image

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

After the driver has been loaded, you can establish a connection to the data source by using the `DriverManager` class in the `java.sql` package.

The `getConnection(String, String, String)` method of `DriverManager` can be used to set up the connection. It returns a reference to a `Connection` object representing an active data connection.

This method has three arguments:

- A string identifying the data source and the type of database connectivity used to reach it
- A username
- A password

The last two items are needed only if the data source is secured with a username and password. If it isn't, these arguments can be null strings ("").

Here's the string to use when connecting to the `sample` database:

Click here to view code image

```
jdbc:derby://localhost:1527/sample
```

You've already seen this string in the Services tab of the Projects pane, where it is an item that represents a database connection.

This string identifies the type of database (jdbc:derby:), the host and port of the database server (localhost:1527), and the name of the database (sample). Note the two slash characters (//) after the database type and the one slash after the host and port.

The second and third arguments to use are app and APP, capitalized as shown. They're the username and password.

The following statement could be used to connect to a database called payroll with a username of doc and a password of 1rover1:

Click here to view code image

```
Connection payday = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/payroll",
    "doc", "1rover1");
```

After you have a connection, you can reuse it each time you want to retrieve information from or store information to that connection's data source.

The `getConnection()` method and all others called on a data source throw `SQLException` errors if something goes wrong as the data source is being used. SQL has its own error messages, and they are passed along as part of `SQLException` objects.

Tip

NetBeans shows the information required to connect to a database, including the driver class, database connection string, username, and password. Right-click the database connection, such as `jdbc:derby://localhost:1527/sample`, and choose Properties from the pop-up menu. A dialog containing the class and other information about the connection appears.

A SQL statement is represented in Java by a `Statement` object. `Statement` is an interface, so it can't be instantiated directly. However, an object that implements the interface is returned by the `createStatement()` method of a `Connection` object, as in the following example:

Click here to view code image

```
Statement lookSee = payday.createStatement();
```

After you have a `Statement` object, you can use it to conduct a SQL query by calling the object's `executeQuery(`*`String`*`)` method. The *`String`* argument should be a SQL query that follows the syntax of that language.

Caution

It's beyond the scope of today's lesson to teach SQL, a rich data-retrieval and storage language that has its own new book from this publisher: *Sams Teach Yourself SQL in 24 Hours*, 6th Edition, by Ryan Stephens, Arie D. Jones, and Ron Plew (ISBN: 0-672-33759-2). Although you need to learn SQL to do extensive work with it, much of the language is easy to pick up from any examples you can find, such as those you will work with today.

The following is an example of a SQL query that could be used on the `sample` database:

Click here to view code image

```
select NAME, CITY from APP.CUSTOMER where (STATE = 'FL')
    order by CITY;
```

This SQL query retrieves several fields for each record in the database for which the `STATE` field equals "FL". The records returned are sorted according to their `CITY` field. The lowercase parts of the command are SQL keywords. The uppercase parts are aspects of the table.

The following Java statement executes that query on a `Statement` object named `looksee`:

Click here to view code image

```
ResultSet set = looksee.executeQuery(
    "select NAME, CITY from APP.CUSTOMER "
    + " where (STATE = 'FL') order by CITY";
);
```

Although SQL queries end with a semicolon character (;), one is not needed in the argument to `executeQuery()`.

If the SQL query has been phrased correctly, the `executeQuery()` method returns a `ResultSet` object holding all the records that have been retrieved from the data source.

Note

To add records to a database instead of retrieving them, you should call the statement's `executeUpdate()` method. You'll work with this method later.

When a `ResultSet` is returned from `executeQuery()`, it is positioned at the first record that has been retrieved. The following methods of `ResultSet` can be used to pull information from the current record:

- `getDate(String)` returns the `Date` value stored in the specified field name (using the `Date` class in the `java.sql` package, not `java.util.Date`).
- `getDouble(String)` returns the `double` value stored in the specified field name.
- `getFloat(String)` returns the `float` value stored in the specified field.
- `getInt(String)` returns the `int` value in the field.
- `getLong(String)` returns the `long` value in the field.
- `getString(String)` returns the `String` in the field.

These are just the simplest methods available in the `ResultSet` interface. The methods you should use depends on the form that the field data takes in the database. But methods such as `getString()` and `getInt()` can be more flexible in the information they retrieve from a record.

You also can use an integer as the argument to any of these methods, such as `getString(5)`, instead of a string. The integer indicates which field to retrieve (1 for the first field, 2 for the second field, and so on).

A `SQLException` is thrown if a database error occurs as you try to retrieve information from a resultset. You can call this exception's `getSQLState()` and `getErrorCode()` methods to learn more about the error.

After you have pulled the information you need from a record, you can move to the next record by calling the `next()` method of the `ResultSet` object. This method returns a `false` Boolean value when it tries to move past the end of a resultset.

Normally, you can move through a resultset once from start to finish, after which you can't retrieve its contents again.

When you're finished using a connection to a data source, you can close it by calling the connection's `close()` method with no arguments.

Listing 18.1 presents the CustomerReporter application, which uses the Java DB driver and a SQL statement to retrieve records from a table in the `sample` database. Four fields are retrieved from each record indicated by the SQL statement: `TABLEID`, `TABLENAME`, `TABLETYPE`, and `SCHEMAID`. The resultset is sorted according to the `TABLENAME` field, and these fields are displayed.

Before creating this application, you must add the `JavaDB` library to the project in NetBeans:

1. Click the Projects tab in the Projects pane to bring it to the front.
2. Scroll down to the bottom of the pane and right-click the `Libraries` folder.
3. Click Add Library from the pop-up menu that appears. The Add Library dialog opens.
4. Choose `JavaDB` under Available Libraries and click Add Library.

The library now appears in the Libraries folder. Three new JAR files will appear in the Libraries item in the Projects pane: `derby.jar`, `derbyclient.jar`, and `derbynet.jar`. The driver necessary to

access the sample database on the Java DB server will be available to the CustomerReporter application.

Create the `CustomerReporter` class in the `com.java21days` package in NetBeans with the source code of the listing.

LISTING 18.1 The Full Text of `CustomerReporter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.sql.*;
 4:
 5: public class CustomerReporter {
 6:     public static void main(String[] arguments) {
 7:         String data = "jdbc:derby://localhost:1527/sample";
 8:         try (
 9:             Connection conn = DriverManager.getConnection(
10:                 data, "app", "APP");
11:             Statement st = conn.createStatement()) {
12:
13:             Class.forName("org.apache.derby.jdbc.ClientDriver"
);
14:
15:              ResultSet rec = st.executeQuery(
16:                  "select CUSTOMER_ID, NAME, CITY, STATE " +
17:                  "from APP.CUSTOMER " +
18:                  "order by CUSTOMER_ID");
19:              while (rec.next()) {
20:                  System.out.println("CUSTOMER_ID:\t"
21:                      + rec.getString(1));
22:                  System.out.println("NAME:\t" +
rec.getString(2));
23:                  System.out.println("CITY:\t" +
rec.getString(3));
24:                  System.out.println("STATE:\t" +
rec.getString(4));
25:                  System.out.println();
26:              }
27:              st.close();
28:         } catch (SQLException s) {
29:             System.out.println("SQL Error: " + s.toString() +
" "
30:                 + s.getErrorCode() + " " + s.getSQLState());
31:         } catch (Exception e) {
32:             System.out.println("Error: " + e.toString()
33:                 + e.getMessage());
34:         }
```

```
35:         }
36: }
```

When this program is run with the starting data from the `sample` database, part of the output is shown in Figure 18.6.

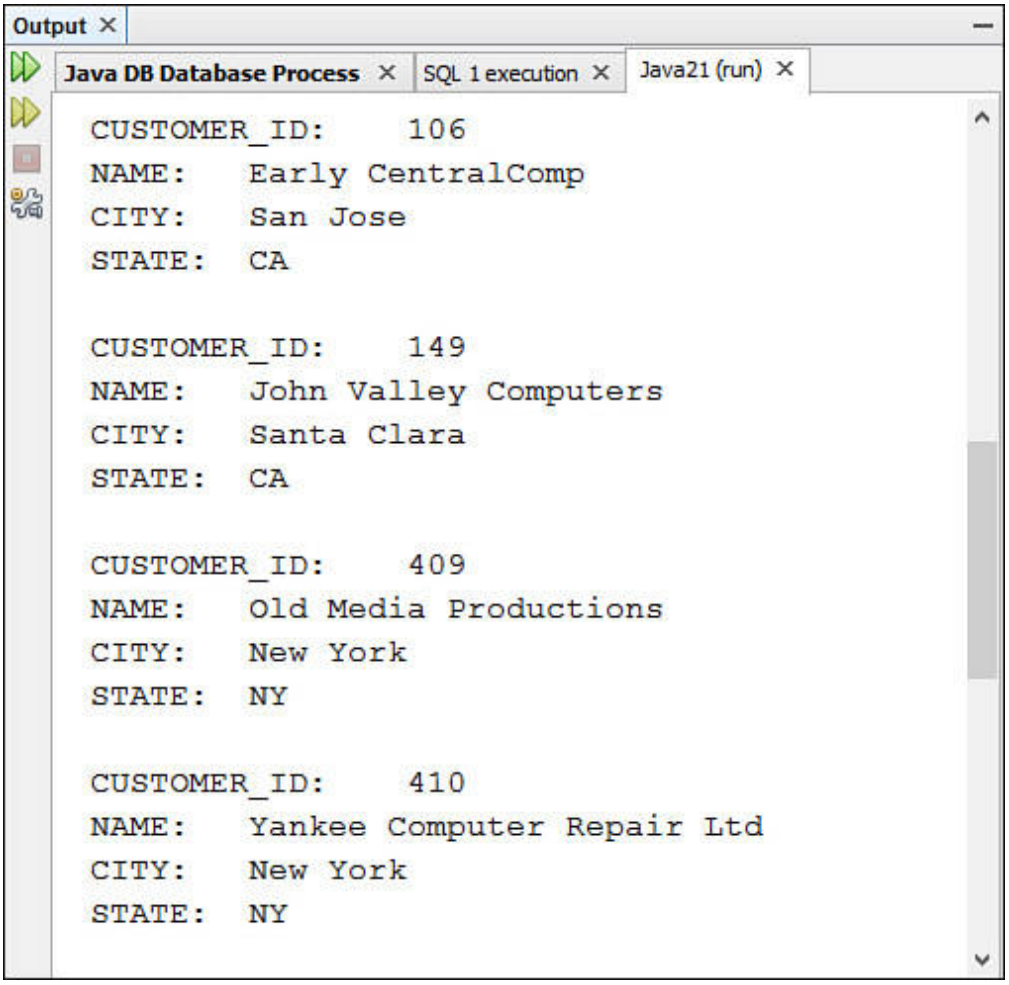


FIGURE 18.6 Reading records from a Java DB database.

Caution

If you run this application and it fails with an SQL error stating "Connection authentication failure occurred. Reason: Userid or password invalid," it may be due to a bug in NetBeans. Change the password used as the final argument to the `getConnection()` method in Lines 9–10 from "APP" to "app" and run the program again to see if it resolves the problem.

Writing Records to a Database

In the CustomerReporter application, you retrieved data from a database using a SQL statement prepared as a string:

Click here to view code image

```
select CUSTOMER_ID, NAME, CITY, STATE from APP.CUSTOMER
    order by CUSTOMER_ID;
```

This is a common way to use SQL. You could write a program that asks a user to enter a SQL query and then displays the result. (However, this would be a terrible idea, because SQL queries can be used to delete records, tables, and even entire databases.)

The `java.sql` package also supports another way to create a SQL statement: a prepared statement.

A prepared statement, which is represented by the `PreparedStatement` class, is a SQL statement that is compiled before it is executed. This enables the statement to return data more quickly and is a better choice if you are executing a SQL statement repeatedly in the same program.

To create a prepared statement, call a connection's `prepareStatement(`*`String`*`)` method with a string that indicates the structure of the SQL statement.

To indicate the structure, you write a SQL statement in which parameters have been replaced with question marks.

Here's an example for a connection object called `cc`:

Click here to view code image

```
PreparedStatement ps = cc.prepareStatement(
    "select * from APP.CUSTOMER where (ZIP=?) "
    + "order by NAME");
```

Here's another example with more than one question mark:

Click here to view code image

```
PreparedStatement ps = cc.prepareStatement(
    "insert into APP.CUSTOMER " +
    "VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
```

The question marks in these SQL statements are placeholders for data. Before you can execute the statement, you must put data in each of these places using one of the methods of the `PreparedStatement` class.

To put data into a prepared statement, you must call a method with the position of the placeholder followed by the data to insert.

For example, to put the string "Acme Corp." in the fifth field of the prepared statement, call the `setString(int, String)` method:

```
ps.setString(5, "Acme Corp.");
```

The first argument indicates the placeholder's position, numbered from left to right. The first question mark is 1, the second is 2, and so on.

The second argument is the data to put in the statement at that position.

The following methods are available:

- `setAsciiStream(int, InputStream, int)`—At the position indicated by the first argument, insert the specified `InputStream`, which represents a stream of ASCII characters. The third argument indicates how many bytes from the input stream to insert.
- `setBinaryStream(int, InputStream, int)`—At the position indicated by the first argument, insert the specified `InputStream`, which represents a stream of bytes. The third argument indicates how many bytes to insert from the stream.
- `setCharacterStream(int, Reader, int)`—At the position indicated by the first argument, insert the specified `Reader`, which represents a character stream. The third argument indicates how many characters to insert from the stream.
- `setBoolean(int, boolean)`—Inserts a `boolean` value at the position indicated by the integer.
- `setByte(int, byte)`—Inserts a `byte` value at the indicated position.
- `setBytes(int, byte[])`—Inserts an array of bytes at the indicated position.
- `setDate(int, Date)`—Inserts a `Date` object (from the `java.sql` package) at the indicated position.
- `setDouble(int, double)`—Inserts a `double` value at the indicated position.
- `setFloat(int, float)`—Inserts a `float` value at the indicated position.
- `setInt(int, int)`—Inserts an `int` value at the indicated position.

- `setLong(int, long)`—Inserts a `long` value at the indicated position.
- `setShort(int, short)`—Inserts a `short` value at the indicated position.
- `setString(int, String)`—Inserts a `String` value at the indicated position.

There's also a `setNull(int, int)` method that stores SQL's version of a null (empty) value at the position indicated by the first argument.

The second argument to `setNull()` should be a class variable from the `Types` class in `java.sql` to indicate what kind of SQL value belongs in that position.

There are class variables for each of the SQL data types. This list, which is not complete, includes some of the most commonly used variables: `BIGINT`, `BIT`, `CHAR`, `DATE`, `DECIMAL`, `DOUBLE`, `FLOAT`, `INTEGER`, `SMALLINT`, `TINYINT`, and `VARCHAR`.

The following code puts a null `CHAR` value at the fifth position in a prepared statement called `ps`:

```
ps.setNull(5, Types.CHAR);
```

The next project demonstrates the use of a prepared statement to add stock quote data to a database. Quotes are collected from Yahoo!.

As a service to people who follow the stock market, Yahoo! offers a Download Spreadsheet link on its main stock quote page for each ticker symbol.

To see this link, look up a stock quote on Yahoo! or go directly to a page such as this one:

Click here to view code image

```
http://quote.yahoo.com/q?s=fb&d=v1
```

At the bottom of the page under the Toolbox heading, you can find a Download Data link. Here's what the link to Facebook looks like:

Click here to view code image

```
http://download.finance.yahoo.com/d/quotes.csv?
s=FB&f=sl1d1t1c1ohgv&e=.csv
```

You can click this link to open the file or save it to a folder on your system. The file, which is only one line long, contains the stock's price and volume data saved at the last market close. Here's an example of what Facebook's data looked like on Sept. 25, 2015:

Click here to view code image

```
"FB",92.77,"9/25/2015","4:00pm",-1.64,95.85,95.85,92.06,28961622
```

The fields in this data, in order, are the ticker symbol, closing price, date, time, price change since yesterday's close, daily low, daily high, daily open, and volume.

The QuoteData application uses each of these fields except one—the time, which isn't particularly useful because it's always the time the market closed.

The following takes place in the program:

- A stock's ticker symbol is used as a command-line argument.
- A `QuoteData` object is created with the ticker symbol as an instance variable called `ticker`.
- The object's `retrieveQuote()` method is called to download the stock data from Yahoo! and return it as a `String`.
- The object's `storeQuote()` method is called with that `String` as an argument. It saves the stock data to a database using a JDBC-ODBC connection.

Before you can run the application, you must have a database table designed to hold these stock quotes.

You can create a new table for this purpose in the `sample` database in NetBeans by following these steps:

1. In the Services tab of the Projects pane, open the APP item under the jdbc:derby://localhost:1527/sample item.
2. Right-click this item's Tables folder and choose Create Table from the pop-up menu. The Create Table dialog opens, as shown in Figure 18.7.

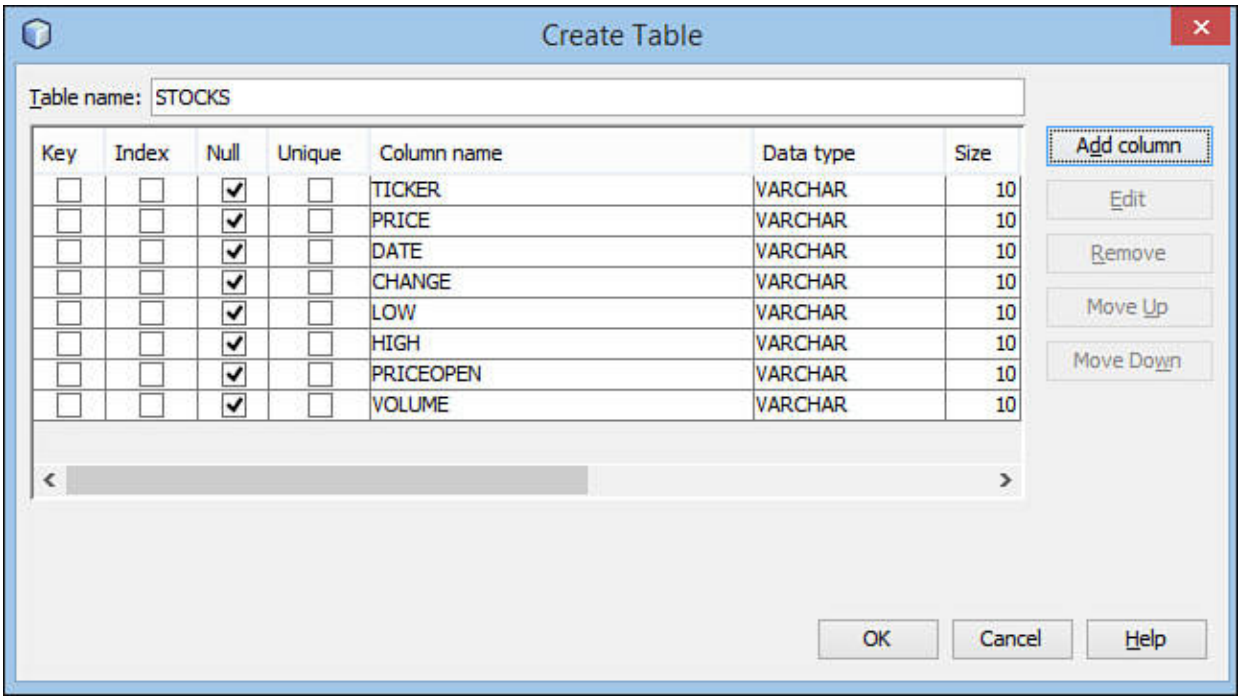


FIGURE 18.7 Creating a new database table in NetBeans.

3. In the Table Name field, enter `STOCKS`.
4. Click Add column. The Add Column dialog opens.
5. In the Name field, enter `TICKER`.
6. In the Type field, choose VARCHAR.
7. In the Size field, enter `10`.
8. Click OK. The new field appears in the dialog.
9. Repeat steps 4–8 for fields named `PRICE`, `DATE`, `CHANGE`, `LOW`, `HIGH`, `PRICEOPEN`, and `VOLUME`. The type and size are always `VARCHAR` and `10`, respectively.
10. Click OK. The STOCKS table appears in the Tables folder.

Now that you have a database table, you can create the QuoteData application, shown in Listing 18.2, to store stock data in a new record of that table. Create the class `QuoteData` in the `com.java21days` package in NetBeans.

LISTING 18.2 The Full Text of `QuoteData.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.sql.*;
 6: import java.util.*;
 7:
 8: public class QuoteData {
 9:     private String ticker;
10:
11:     public QuoteData(String inTicker) {
12:         ticker = inTicker;
13:     }
14:
15:     private String retrieveQuote() {
16:         StringBuilder builder = new StringBuilder();
17:         try {
18:             URL page = new URL(
19:                 "http://quote.yahoo.com/d/quotes.csv?s=" +
20:                 ticker + "&f=sl1d1t1c1ohgv&e=.csv");
21:             String line;
22:             URLConnection conn = page.openConnection();
23:             conn.connect();
24:             InputStreamReader in = new InputStreamReader(
25:                 conn.getInputStream());
26:             BufferedReader data = new BufferedReader(in);
27:             while ((line = data.readLine()) != null) {
28:                 builder.append(line);
29:                 builder.append("\n");
30:             }
31:         } catch (MalformedURLException mue) {
32:             System.out.println("Bad URL: " +
mue.getMessage());
33:         } catch (IOException ioe) {
34:             System.out.println("IO Error:" +
ioe.getMessage());
35:         }
36:         return builder.toString();
37:     }
38:
39:     private void storeQuote(String data) {
40:         StringTokenizer tokens = new StringTokenizer(data,
",");
41:         String[] fields = new String[9];
42:         for (int i = 0; i < fields.length; i++) {
43:             fields[i] = stripQuotes(tokens.nextToken());
44:         }
45:         String datasource =
"jdbc:derby://localhost:1527/sample";
46:         try (
47:             Connection conn = DriverManager.getConnection(
```

```
48:                 datasource, "app", "app")
49:             ) {
50:
51:             Class.forName("org.apache.derby.jdbc.ClientDriver
");
52:             PreparedStatement prep2 = conn.prepareStatement(
53:                 "insert into " +
54:                 "APP.STOCKS(TICKER, PRICE, DATE, CHANGE, LOW,
" +
55:                 "HIGH, PRICEOPEN, VOLUME) " +
56:                 "values(?, ?, ?, ?, ?, ?, ?, ?)");
57:             prep2.setString(1, fields[0]);
58:             prep2.setString(2, fields[1]);
59:             prep2.setString(3, fields[2]);
60:             prep2.setString(4, fields[4]);
61:             prep2.setString(5, fields[5]);
62:             prep2.setString(6, fields[6]);
63:             prep2.setString(7, fields[7]);
64:             prep2.setString(8, fields[8]);
65:             prep2.executeUpdate();
66:             prep2.close();
67:             conn.close();
68:         } catch (SQLException sqe) {
69:             System.out.println("SQL Error: " +
sqe.getMessage());
70:         } catch (ClassNotFoundException cnfe) {
71:             System.out.println(cnfe.getMessage());
72:         }
73:     }
74:
75:     private String stripQuotes(String input) {
76:         StringBuilder output = new StringBuilder();
77:         for (int i = 0; i < input.length(); i++) {
78:             if (input.charAt(i) != '\"') {
79:                 output.append(input.charAt(i));
80:             }
81:         }
82:         return output.toString();
83:     }
84:
85:     public static void main(String[] arguments) {
86:         if (arguments.length < 1) {
87:             System.out.println("Usage: java QuoteData
ticker");
88:             System.exit(0);
89:         }
90:         QuoteData qd = new QuoteData(arguments[0]);
91:         String data = qd.retrieveQuote();
92:         qd.storeQuote(data);
93:     }
94: }
```

Before you run the application, you must set a command-line argument. Choose Run, Set Project Configuration, Customize, and then enter the main class `QuoteData` and the argument of a valid ticker symbol, such as `FB` (Facebook), `GOOG` (Google), or `PSO` (Pearson PLC).

The application stores the stock data but does not display any output.

To see that it worked, right-click the `STOCKS` table in the Services tab and choose View Data. The table records are displayed; they should include at least one day's data for the requested stock ticker symbol, as shown in Figure 18.8.

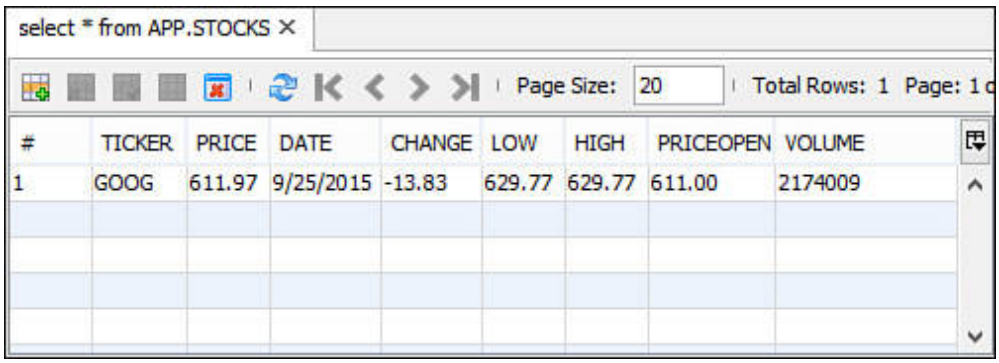


FIGURE 18.8 Records in the `STOCKS` table.

The `retrieveQuote()` method (lines 15–37) downloads the quote data from Yahoo! and saves it as a string. The techniques used in this method were covered on Day 17, "Communicating Across the Internet."

The `storeQuote()` method (lines 39–73) uses the SQL techniques covered in this section.

The method begins by using the `StringTokenizer` class to split the quote data into a set of tokens, using the comma character (,) as the delimiter between each token. The tokens then are stored in a `String` array with nine elements.

The array contains the same fields as the Yahoo! data in the same order: ticker symbol, closing price, date, time, price change, low, high, open, and volume.

Next, a data connection to the `QuoteData` data source is created using the Java DB database driver (lines 45–49).

This connection then is used to create a prepared statement (lines 52–56). This statement uses the `insert into` SQL statement, which causes data to be stored in a database. In this case, the database is `sample`, and the `insert into` statement refers to the `APP.STOCKS` table in that database.

The prepared statement has eight placeholders. Only eight are needed, instead of nine, because the application does not use the time field from the Yahoo! data.

A series of `setString()` methods puts the elements of the `String` array into the prepared statement, in the same order that the fields exist in the database: ticker symbol, closing price, date, price change, low, high, open, and volume (lines 57–64).

Because some fields in the Yahoo! data are dates, floating-point numbers, and integers, you might think that it would be better to use `setDate()`, `setFloat()`, and `setInt()` for that data. This application stores all the stock data as strings because that's more likely to work regardless of the database software being used.

Caution

Some databases you could use in Java programs, including Microsoft Access, do not support some of these methods when you are using SQL to work with the database, even though they exist in Java. If you try to use an unsupported method, such as `setFloat()`, a `SQLException` error occurs.

It's easier to send a database strings and let the database program automatically convert them into the correct format. This is likely to be true when you are working with other databases; the level of SQL support varies based on the product and driver involved.

After the statement has been prepared and all the placeholders are filled, the statement's `executeUpdate()` method is called in line 65. This either adds the quote data to the database or throws a SQL error.

The private method `stripQuotes()` is used to remove quotation marks from Yahoo!'s stock data. This method is called in line 43 to take care of three fields that contain extraneous quotes: the ticker symbol, date, and time.

Moving Through Resultsets

The default behavior of resultsets permits one trip through the set using its `next()` method to retrieve each record.

By changing how statements and prepared statements are created, you can produce resultsets that support these additional methods:

- `afterLast()` moves to a place immediately after the last record in the set.
- `beforeFirst()` moves to a place immediately before the first record in the set.
- `first()` moves to the first record in the set.
- `last()` moves to the last record in the set.
- `previous()` moves to the previous record in the set.

These actions are possible when the resultset's policies have been specified as arguments to a database connection's `createStatement()` and `prepareStatement()` methods.

Normally, `createStatement()` takes no arguments, as in this example:

Click here to view code image

```
Connection payday = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/sample", "Doc", "1rover1");
Statement lookSee = payday.CreateStatement();
```

For a more flexible resultset, call `createStatement()` with three integer arguments that set up how it can be used. Here's a rewrite of the preceding statement:

Click here to view code image

```
Statement lookSee = payday.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

The same three arguments can be used in the `prepareStatement(String, int, int, int)` method after the text of the statement.

The `ResultSet` class includes other class variables that offer more options in how sets can be read and modified.

Summary

Today you learned to read and write database records using classes that work with any of the popular relational databases. The techniques used to work with Java DB can be used with Microsoft Access, MySQL, and other programs. The only thing that needs to be changed is the database driver class and the strings used to create a connection.

Using Java Database Connectivity (JDBC), you can incorporate existing data-storage solutions into your Java programs.

You can connect to several different relational databases in your Java programs by using JDBC and Structured Query Language (SQL), a standard language for reading, writing, and managing a database.

Q&A

Q What's the difference between Java DB and more well-known databases such as Access and MySQL? Which should I use?

A Java DB is intended for database applications that have simpler needs than Access and comparable databases. The entire application takes up under 4MB of space, making it easy to bundle with Java applications that require database connectivity.

Oracle employs Java DB in several parts of the Java Enterprise Edition, which demonstrates that it can deliver strong, reliable performance on important tasks.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What does a `Statement` object represent in a database program?

   A. A connection to a database

   B. A database query written in Structured Query Language

   C. A data source

2. Which Java class represents SQL statements that are compiled before they are executed?

   A. `Statement`

B. `PreparedStatement`

C. `ResultSet`

3. What does the `Class.forName(`*`String`*`)` method accomplish?

A. It provides the name of a class.

B. It loads a database driver that can be used to access a database.

C. It deletes an object.

Answers

1. B. The class, part of the `java.sql` package, represents a SQL statement.
2. B. Because it is compiled, `PreparedStatement` is a better choice when you need to execute the same SQL query numerous times.
3. B. This static method loads a database driver.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class ArrayClass {

    public static ArrayClass newInstance() {
        count++;
        return new ArrayClass();
    }

    public static void main(String arguments[]) {
        new ArrayClass();
    }

    int count = -1;
}
```

Which line in this program prevents it from compiling successfully?

A. `count++;`

B. `return new ArrayClass();`

C. `public static void main(String arguments[]) {`

D. `int count = -1;`

The answer is available on the book's website at www.java21days.com. Visit the Day 18 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Modify the CustomerReporter application to pull fields from another table in `APP`.
2. Write an application that reads and displays records from the Yahoo! stock quote database.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 19. Reading and Writing RSS Feeds

Today, you work with Extensible Markup Language (XML), a popular and widely implemented formatting standard that enables data to be portable.

You explore XML in the following ways:

- Representing data as XML
- Discovering why XML is a useful way to store data
- Using XML to publish web content
- Reading and writing XML data

The XML format employed throughout the day is Really Simple Syndication (RSS), a popular way to publish web content and share information on site updates. RSS has been adopted by millions of sites.

Using XML

One of Java's main selling points is that the language produces programs that can run on different operating systems without modification. The portability of software is a big convenience in today's computing world, where Windows, Linux, Mac OS, iOS, Android, and other operating systems are in wide use and many people work with multiple systems.

XML is a format for storing and organizing data that is independent of any software program that works with the data.

Data that is compliant with XML is easier to reuse for several reasons.

First, the data is structured in a standard way, making it possible for software to read and write the data as long as it supports XML. If you create an XML file that represents your company's employee database, several dozen XML parsers can read the file and make sense of its contents.

This is true no matter what kind of information you collect about each employee. If your database contains only the employee's name, ID number, and salary, XML parsers can read it. If it contains 25 items, including birthday, blood type, and hair color, parsers can read that, too.

Second, the data is self-documenting, making it easier for people to understand a file's purpose by looking at it in a text editor. Anyone who opens

your XML employee database should be able to figure out the structure and content of each employee record without any assistance from you.

This is evident in Listing 19.1, which contains an RSS file. Because RSS is an XML dialect, it is structured under the rules of XML. Enter this code in NetBeans (category Other, type Empty File) and save it as `workbench.rss`. (You also can download a copy of it from the book's website at www.java21days.com on the Day 19 page.)

LISTING 19.1 The Full Text of `workbench.rss`

Click here to view code image

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <rss version="2.0">
 3:   <channel>
 4:     <title>Workbench</title>
 5:     <link>http://workbench.cadenhead.org/</link>
 6:     <description>Programming, publishing, and
popes</description>
 7:     <docs>http://www.rssboard.org/rss-specification</docs>
 8:     <item>
 9:       <title>Programming Confidence Pool for the World
Cup</title>
10:       <link>http://workbench.cadenhead.org/news/739</link>
11:       <pubDate>Wed, 11 Jun 2015 11:49:47 -0400</pubDate>
12:       <guid
isPermaLink="false">tag:cadenhead.org,2015:w.739</guid>
13:       <enclosure length="2498623" type="audio/mpeg"
14:           url="http://mp3.cadenhead.org/3679.mp3" />
15:     </item>
16:     <item>
17:       <title>Ghost of Computer Author Past</title>
18:       <link>http://workbench.cadenhead.org/news/737</link>
19:       <pubDate>Mon, 24 Mar 2014 17:00:13 -0400</pubDate>
20:       <guid
isPermaLink="false">tag:cadenhead.org,2015:w.737</guid>
21:     </item>
22:     <item>
23:       <title>Interview with Zoe Zolbrod</title>
24:       <link>http://workbench.cadenhead.org/news/736</link>
25:       <pubDate>Fri, 21 Mar 2014 11:12:55 -0400</pubDate>
26:       <guid
isPermaLink="false">tag:cadenhead.org,2015:w.736</guid>
27:     </item>
28:   </channel>
29: </rss>
```

Can you tell what the data represents? Although the `?xml` tag at the top might be indecipherable, the rest is clearly a website database of some kind.

The `?xml` tag in the first line of the file has a `version` attribute with a value of "1.0" and an `encoding` attribute of "utf-8". This establishes that the file follows the rules of XML 1.0 and is encoded with the UTF-8 character set.

Data in XML is surrounded by tag elements that describe the data. Opening tags begin with a < character followed by the name of the tag and a > character. Closing tags begin with the `</` characters followed by a name and a > character. In Listing 19.1, for example, `<item>` on line 8 is an opening tag, and `</item>` on line 15 is a closing tag. Everything within those tags is considered to be the value of that element.

Elements can be nested within other elements, creating a hierarchy of XML data that establishes relationships within that data. In Listing 19.1, everything in lines 9–14 is related; each element defines something about the same website item.

Elements also can include attributes, which are made up of data that supplements the rest of the data associated with the element. Attributes are defined within an opening tag element. The name of an attribute is followed by an equal sign and text within quotation marks.

In line 12 of Listing 19.1, the `guid` element includes an `isPermaLink` attribute with a value of "false". This indicates that the element's value, tag:cadenhead.org,2015:w.739, is not a permalink, the URL at which the item can be loaded in a browser.

XML also supports elements defined by a single tag rather than a pair of tags. The tag begins with a < character followed by the name of the tag and ends with the `/>` characters. The RSS file includes an `enclosure` element in lines 13–14 that describes an MP3 audio file associated with the item.

XML encourages the creation of data that's understandable and usable even if the user doesn't have the program that created it and cannot find any documentation that describes it.

For the most part, you can understand the purpose of the RSS file shown in Listing 19.1 simply by looking at it. Each item represents a web page that has been updated recently.

Tip

Publishing new site content with RSS and a similar format, Atom, has become one of the best ways to build readership on the Web. Thousands of people subscribe to RSS files, which are called feeds, using reader software such as Feedly and My Yahoo!.

Rogers Cadenhead, the author of this book, is the chairman of the RSS Advisory Board, the group that publishes the RSS 2.0 specification. For more information on the format, visit the board's website at www.rssboard.org or subscribe to its RSS feed at www.rssboard.org/rss-feed.

There's also another version of RSS, RDF Site Summary, that's used on some sites for its feeds. Find out more at http://web.resource.org/rss/1.0.

Data that follows XML's formatting rules is said to be well-formed. Any software that can work with XML reads and writes well-formed XML data.

By insisting on well-formed markup, XML simplifies the task of writing programs that work with the data. RSS makes website updates available in a form that software can easily process. The RSS feed for Workbench at http://feeds.cadenhead.org/workbench has two distinct audiences: humans reading the blog through their preferred RSS reader, and computers that do something with this data. Twitter, Facebook, and many other sites can pull data from an RSS feed and present it to users.

Designing an XML Dialect

Although XML is described as a language and is compared to Hypertext Markup Language (HTML), it's actually much larger in scope. XML is a markup language that defines how to define a markup language.

That's an odd distinction to make, and probably sounds like something you'd encounter in a philosophy textbook. This concept is important to understand because it explains how XML can be used to define data as varied as health-care claims, genealogical records, newspaper articles, and molecules.

The X in XML stands for Extensible, and it refers to organizing data for your own purposes. Data that's organized using the rules of XML can represent anything you want:

- A programmer at a telemarketing company can use XML to store data on each outgoing call, saving the time of the call, the number, the operator who made the call, and the result.
- A hobbyist can use XML to keep track of the annoying telemarketing calls she receives, noting the time of the call, the company, and the product being peddled.
- A programmer at a government agency can use XML to track complaints about telemarketers, saving the name of the marketing firm and the number of complaints.

Each of these examples uses XML to define a new language that suits a specific purpose. Although you could call them XML languages, they're more commonly described as XML dialects or XML document types.

An XML dialect can be designed using a document type definition (DTD) that indicates the potential elements and attributes it covers.

A special `!DOCTYPE` declaration can be placed in XML data, right after the initial `?xml` tag, to identify its DTD. Here's an example:

Click here to view code image

```
<!DOCTYPE Library SYSTEM "librml.dtd">
```

The `!DOCTYPE` declaration is used to identify the DTD that applies to the data. When a DTD is present, many XML tools can read XML created for that DTD and determine whether the data follows all the rules. If it doesn't, it is rejected with a reference to the line that caused the error. This process is called *validating the XML*.

One thing you run into as you work with XML is data that has been structured as XML but wasn't defined using a DTD. Most versions of RSS files do not require a DTD. This data can be parsed (presuming it's well-formed), so you can read it into a program and do something with it, but you can't check its validity to make sure that it's organized correctly according to the rules of its dialect.

Tip

To give you an idea of what kinds of XML dialects have been created, Cover Pages offers a list at http://xml.coverpages.org/xmlApplications.html.

Processing XML with Java

Java supports XML through the Java API for XML Processing, a set of packages for reading, writing, and manipulating XML data.

The `javax.xml.parsers` package is the entry point to the other packages. These classes can be used to parse and validate XML data using two techniques: the Simple API for XML (SAX) and the Document Object Model (DOM). However, they can be difficult to implement, which has inspired other groups to offer their own class libraries to work with XML.

You spend the remainder of the day working with one of these alternatives: the XML Object Model (XOM) library, an open source Java class library that makes it extremely easy to read, write, and transform XML data.

Note

To find out more about the Java API for XML Processing, visit Oracle's Java website at http://docs.oracle.com/javase/8/docs/technotes/guides/xml.

Processing XML with XOM

One of the most important skills you can develop as a Java programmer is the ability to find suitable packages and classes that can be employed in your own projects. For obvious reasons, using a well-designed class library is much easier than developing one on your own.

Although the Java Class Library contains thousands of well-designed classes that cover a comprehensive range of development needs, Oracle isn't the only supplier of classes that may prove useful to your efforts.

Other companies, groups, and individuals offer dozens of Java packages under a variety of commercial and open source licenses. Some of the most notable come from the Apache Software Foundation, whose Java projects include the web application framework Struts, the Java servlet container Tomcat, and the Log4J logging class library.

Another terrific open source Java class library is the XOM library. This tree-based package for XML processing strives to be simple to learn, easy to use, and uncompromising in its adherence to well-formed XML.

The library was developed by the programmer and author Elliotte Rusty Harold based on his experience with XML processing in Java.

XOM originally was envisioned as a fork of JDOM, a popular tree-based model for representing an XML document. Harold contributed code to that open source project and participated in its development. But instead of forking the JDOM code, Harold decided to start from scratch and adopt some of its core design principles in XOM.

The library embodies the following principles:

- XML documents are modeled as a tree, with Java classes representing nodes on the tree such as elements, comments, processing instructions, and document type definitions. A programmer can add and remove nodes to manipulate the document in memory, a simple approach that can be implemented gracefully in Java.
- All XML data produced by XOM is well-formed and has a well-formed namespace.
- Each element of an XML document is represented as a class with constructors.
- Object serialization is not supported. Instead, programmers are encouraged to use XML as the format for serialized data, enabling it to be readily exchanged with any software that reads XML, regardless of the programming language in which it was developed.
- The library relies on another XML parser to read XML documents and fill trees. XOM uses a SAX parser that must be downloaded and installed separately. Apache Xerces 2.6.1 and later versions should work.

XOM is available for download from www.xom.nu. The current version is 1.2.10, which includes Xerces 2.8 in its distribution.

Caution

XOM is released under the open source GNU Lesser General Public License (LGPL), which grants permission to distribute the library without modification with Java programs that use it.

You also can make changes to the XOM class library as long as you offer them under the LGPL. The full license is published online at www.xom.nu/license.xhtml.

XOM can be downloaded as a ZIP or TAR.GZ archive. Download the library and extract the files on a folder on your computer; then follow these steps to add it to NetBeans:

1. Choose Tools, Libraries. The Ant Library Manager opens.
2. Click New Library. The New Library dialog appears.
3. Enter `XOM 1.2.10` as the Library Name, and click OK.
4. Back in the Ant Library Manager, click Add JAR/Folder.
5. Browse to the folder where you extracted the XOM archive, and open it.
6. In that folder, choose the file `xom-1.2.10.jar`.
7. Click Add JAR/Folder.
8. In the Ant Library Manager, click OK.

After you have added the library to NetBeans, you need to add it to the current project so that you can use XOM classes in your programs:

1. In the Projects pane, scroll down past the `.java` files until you see a folder named Libraries.
2. Right-click this folder and choose Add Library. The Add Library dialog appears.
3. Choose XOM 1.2.10, and click OK.

An item for XOM appears under Libraries in the Projects pane.

Creating an XML Document

The first application you develop today, RssWriter, creates an XML document that contains the start of an RSS feed. The document is shown in Listing 19.2. (You don't have to type in this listing.)

LISTING 19.2 The Full Text of `feed.rss`

Click here to view code image

```
1: <?xml version="1.0"?>
2: <rss version="2.0">
3:   <channel>
4:     <title>Workbench</title>
5:     <link>http://workbench.cadenhead.org/</link>
6:   </channel>
7: </rss>
```

The base `nu.xom` package contains classes for a complete XML document (`Document`) and the nodes a document can contain (`Attribute`, `Comment`, `DocType`, `Element`, `ProcessingInstruction`, and `Text`).

The RssStarter application uses several of these classes. First, an `Element` object is created by specifying the element's name as an argument:

Click here to view code image

```
Element rss = new Element("rss");
```

This statement creates an object for the root element of the document, `rss`. `Element`'s one-argument constructor can be used because the document does not employ a feature of XML called namespaces; if it did, a second argument would be necessary: the element's namespace uniform resource identifier (URI). The other classes in the XOM library support namespaces in a similar manner.

In the XML document in Listing 19.2, the `rss` element includes an attribute named `version` with the value "2.0". An attribute can be created by specifying its name and value in consecutive arguments:

Click here to view code image

```
Attribute version = new Attribute("version", "2.0");
```

Attributes are added to an element by calling its `addAttribute()` method with the attribute as the only argument:

```
rss.addAttribute(version);
```

The text contained within an element is represented by the `Text` class, which is constructed by specifying the text as a `String` argument:

Click here to view code image

```
Text titleText = new Text("Workbench");
```

When an XML document is composed, all its elements end up inside a root element that is used to create a `Document` object—a `Document` constructor is called with the root element as an argument. In the RssStarter application, this element is called `rss`. Any `Element` object can be the root of a document:

Click here to view code image

```
Document doc = new Document(rss);
```

In XOM's tree structure, the classes representing an XML document and its constituent parts are organized into a hierarchy below the generic superclass `nu.xom.Node`. This class has three subclasses in the same package: `Attribute`, `LeafNode`, and `ParentNode`.

To add a child to a parent node, call the parent's `appendChild()` method with the node to add as the only argument. The following code creates two elements—a parent called `channel` and one child element, `link`:

Click here to view code image

```
Element channel = new Element("channel");
Element link = new Element("link");
Text linkText = new Text("http://workbench.cadenhead.org/");
link.appendChild(linkText);
channel.appendChild(link);
```

The `appendChild()` method appends a new child below all other children of that parent. The preceding statements produce this XML fragment:

Click here to view code image

```
<channel>
   <link>http://workbench.cadenhead.org/</link>
</channel>
```

The `appendChild()` method also can be called with a `String` argument instead of a node. A `Text` object representing the string is created and added to the element:

Click here to view code image

```
link.appendChild("http://workbench.cadenhead.org/");
```

After a tree has been created and filled with nodes, it can be displayed by calling the `Document` method `toXML()`, which returns the complete and well-formed XML document as a `String`.

Listing 19.3 shows the complete application. Create the `RssStarter` class in the `com.java21days` package in NetBeans with this listing as the source.

LISTING 19.3 The Full Text of `RssStarter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import nu.xom.*;
 5:
 6: public class RssStarter {
 7:     public static void main(String[] arguments) {
 8:         // create an <rss> element to serve as the document's
root
 9:         Element rss = new Element("rss");
10:
11:         // add a version attribute to the element
12:         Attribute version = new Attribute("version", "2.0");
13:         rss.addAttribute(version);
14:         // create a <channel> element and make it a child of
<rss>
15:         Element channel = new Element("channel");
16:         rss.appendChild(channel);
17:         // create the channel's <title>
18:         Element title = new Element("title");
19:         Text titleText = new Text("Workbench");
20:         title.appendChild(titleText);
21:         channel.appendChild(title);
22:         // create the channel's <link>
23:         Element link = new Element("link");
24:         Text lText = new
Text("http://workbench.cadenhead.org/");
25:         link.appendChild(lText);
```

```
26:             channel.appendChild(link);
27:
28:             // create a new document with <rss> as the root
element
29:             Document doc = new Document(rss);
30:
31:             // Save the XML document
32:             try (
33:                 FileWriter fw = new FileWriter("feed.rss");
34:                 BufferedWriter out = new BufferedWriter(fw);
35:             ) {
36:                 out.write(doc.toXML());
37:             } catch (IOException ioe) {
38:                 System.out.println(ioe.getMessage());
39:             }
40:             System.out.println(doc.toXML());
41:         }
42: }
```

The RssStarter application displays the XML document it creates on standard output and saves it to a file called `feed.rss`. The output is shown in Figure 19.1.

FIGURE 19.1 Creating an XML document with XOM.

XOM automatically precedes a document with an XML declaration.

As you can see in Figure 19.1, the XML produced by this application contains no indentation; elements are stacked on the same line.

Caution

XOM preserves significant white space only when representing XML data. The spaces between elements in the RSS feed contained in Listing 19.2 are strictly for presentation purposes and are not produced automatically when XOM creates an XML document. A subsequent example demonstrates how to control indentation.

Modifying an XML Document

The next project, the DomainEditor application, makes several changes to the XML document that was just produced by the RssStarter application, `feed.rss`. The text enclosed by the `link` element is changed, and a new `item` element is added:

Click here to view code image

```
<item>
  <title>Free the Bound Periodicals</title>
</item>
```

Using the `nu.xom` package, XML documents can be loaded into a tree from several sources: a `File`, `InputStream`, `Reader`, or URL (which is specified as a `String` instead of a `java.net.URL` object).

The `Builder` class represents a SAX parser that can load an XML document into a `Document` object. Constructors can be used to specify a particular parser or to let XOM use the first available parser from this list: Xerces 2, Crimson, Piccolo, GNU Aelfred, Oracle, XP, Saxon Aelfred, or Dom4J Aelfred. If none of these is found, the parser specified by the system property `org.xml.sax.driver` is used. Constructors also determine whether the parser is validating or nonvalidating.

The `Builder()` and `Builder(true)` constructors both use the default parser—most likely a version of Xerces. The presence of the Boolean argument `true` in the second constructor configures the parser to be validating. It would be nonvalidating otherwise. A validating parser throws a `nu.xom.ValidityException` if the XML document doesn't validate according to the rules of its document type definition.

The `Builder` object's `build()` method loads an XML document from a source and returns a `Document` object:

Click here to view code image

```
Builder builder = new Builder();
File xmlFile = new File("feed.rss");
Document doc = builder.build(xmlFile);
```

These statements load an XML document from the file `feed.rss` barring one of two problems: A `nu.xom.ParseException` is thrown if the file does not contain well-formed XML, and a `java.io.IOException` is thrown if the input operation fails.

Elements are retrieved from the tree by calling a method of their parent node.

A `Document` object's `getRootElement()` method returns the document's root element:

Click here to view code image

```
Element root = doc.getRootElement();
```

In the XML document `feed.rss`, the root element is `domains`.

Elements with names can be retrieved by calling their parent node's `getFirstChildElement()` method with the name as a `String` argument:

Click here to view code image

```
Element channel = root.getFirstChildElement("channel");
```

This statement retrieves the `channel` element contained in the `rss` element (or `null` if that element could not be found). Like other examples, this is simplified by the lack of a namespace in the document; there also are methods where a name and namespace are arguments.

When several elements within a parent have the same name, the parent node's `getChildElements()` method can be used instead:

Click here to view code image

```
Elements children = channel.getChildElements();
```

The `getChildElements()` method returns an `Elements` object containing each of the elements. This object is a read-only list and does not change automatically if the parent node's contents change after `getChildElements()` is called.

`Elements` has a `size()` method containing an integer count of the elements it holds. This can be used in a loop to cycle through each element in

turn, beginning with the one at position 0. There's a `get()` method to retrieve each element; call it with the integer position of the element to be retrieved:

Click here to view code image

```
for (int i = 0; i < children.size(); i++) {
    Element link = children.get(i);
}
```

This `for` loop cycles through each `child` element of the `channel` element.

Elements without names can be retrieved by calling their parent node's `getChild()` method with one argument: an integer indicating the element's position within the parent node:

Click here to view code image

```
Text linkText = (Text) link.getChild(0);
```

This statement creates the `Text` object for the text "http://workbench.cadenhead.org/" found within the `link` element. `Text` elements always are at position 0 within their enclosing parent.

To work with this text as a string, call the `Text` object's `getValue()` method, as in this statement:

Click here to view code image

```
if
(linkText.getValue().equals("http://workbench.cadenhead.org/"))
    // ...
}
```

The DomainEditor application only modifies a `link` element enclosing the text "http://workbench.cadenhead.org/". The application makes the following changes: The text of the `link` element is deleted, the new text "http://www.cadenhead.org/" is added in its place, and then a new `item` element is added.

A parent node has two `removeChild()` methods to delete a child node from the document. Calling the method with an integer deletes the child at that position:

Click here to view code image

```
Element channel = domain.getFirstChildElement("channel");
Element link = dns.getFirstChildElement("link");
```

```
link.removeChild(0);
```

These statements would delete the `Text` object contained within the channel's first `link` element.

Calling the `removeChild()` method with a node as an argument deletes that particular node. Extending the previous example, the `link` element could be deleted with this statement:

```
channel.removeChild(link);
```

Listing 19.4 shows the source code of the DomainEditor application. Create this class in NetBeans in the `com.java21days` package.

LISTING 19.4 The Full Text of `DomainEditor.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import nu.xom.*;
 5:
 6: public class DomainEditor {
 7:     public static void main(String[] args) throws IOException
{
 8:         try {
 9:             // create a tree from the XML document feed.rss
10:             Builder builder = new Builder();
11:             File xmlFile = new File("feed.rss");
12:             Document doc = builder.build(xmlFile);
13:
14:             // get the root element <rss>
15:             Element root = doc.getRootElement();
16:
17:             // get its <channel> element
18:             Element channel =
root.getFirstChildElement("channel");
19:
20:             // get its <link> elements
21:             Elements children = channel.getChildElements();
22:             for (int i = 0; i < children.size(); i++) {
23:
24:                 // get a <link> element
25:                 Element link = children.get(i);
26:
27:                 // get its text
28:                 Text linkText = (Text) link.getChild(0);
```

```
29:
30:                 // update any link matching a URL
31:                 if (linkText.getValue().equals(
32:                     "http://workbench.cadenhead.org/")) {
33:
34:                     // update the link's text
35:                     link.removeChild(0);
36:                     link.appendChild("http://www.cadenhead.or
g/");
37:                 }
38:             }
39:
40:             // create new elements and attributes to add
41:             Element item = new Element("item");
42:             Element itemTitle = new Element("title");
43:
44:             // add them to the <channel> element
45:             itemTitle.appendChild(
46:                 "Free the Bound Periodicals"
47:             );
48:             item.appendChild(itemTitle);
49:             channel.appendChild(item);
50:
51:             // Save the XML document
52:             try (
53:                 FileWriter fw = new FileWriter("feed2.rss");
54:                 BufferedWriter out = new BufferedWriter(fw);
55:             ) {
56:                 out.write(doc.toXML());
57:             } catch (IOException ioe) {
58:                 System.out.println(ioe.getMessage());
59:             }
60:             System.out.println(doc.toXML());
61:         } catch (ParsingException pe) {
62:             System.out.println("Parse error: " +
pe.getMessage());
63:             pe.printStackTrace();
64:             System.exit(-1);
65:         }
66:     }
67: }
```

The DomainEditor application displays the modified XML document to standard output and saves it to a file named `feeds2.rss`. You can see the program's output in Figure 19.2.

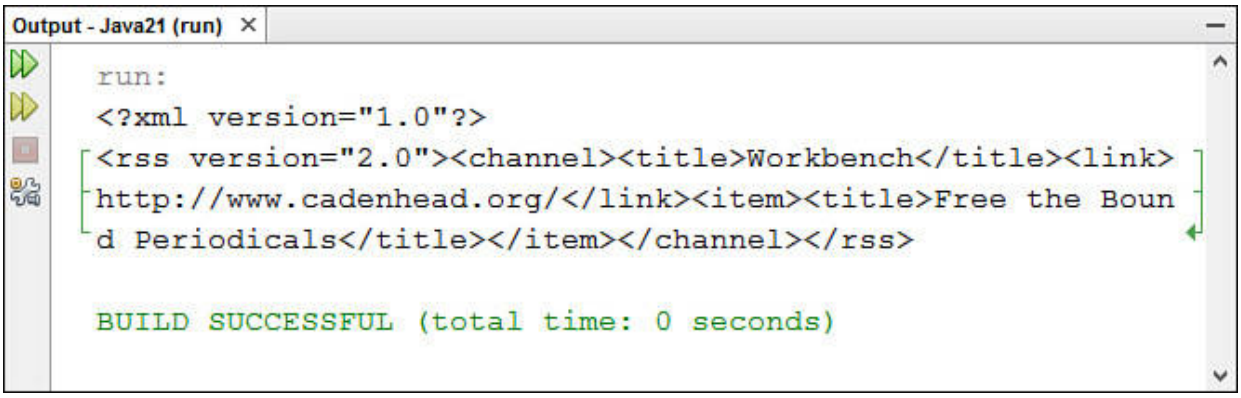


FIGURE 19.2 Loading and modifying an XML document.

Formatting an XML Document

As described earlier, XOM does not retain insignificant white space when representing XML documents. This is in keeping with one of XOM's design goals—to disregard anything that has no syntactic significance in XML. (Another example of this is how text is treated identically whether it is created using character entities, `CDATA` sections, or regular characters.)

Today's next project is the DomainWriter application. This program adds a comment to the beginning of the XML document `feed2.rss` and serializes it with indented lines, producing the version shown in Listing 19.5.

LISTING 19.5 The Full Text of `feed2.rss`

Click here to view code image

```
 1: <?xml version="1.0" encoding="ISO-8859-1"?>
 2: <!--File created Sat Sep 26 23:17:49 EDT 2015-->
 3: <rss version="2.0">
 4:   <channel>
 5:     <title>Workbench</title>
 6:     <link>http://www.cadenhead.org/</link>
 7:     <item>
 8:       <title>Free the Bound Periodicals</title>
 9:     </item>
10:   </channel>
11: </rss>
```

The `Serializer` class in `nu.xom` offers control over how an XML document is formatted when it is displayed or stored serially. Indentation,

character encoding, line breaks, and other formatting are established by objects of this class.

You can create a `Serializer` object by specifying an output stream and character encoding as arguments to the constructor:

Click here to view code image

```
FileOutputStream fos = new FileOutputStream("feed3.rss");
Serializer output = new Serializer(fos, "ISO-8859-1");
```

These statements serialize a file using the ISO-8859-1 character encoding.

Serializer supports 22 encodings, including ISO-10646-UCS-2, ISO-8859-1 through ISO-8859-10, ISO-8859-13 through ISO-8859-16, UTF-8, and UTF-16. There's also a `Serializer()` constructor that takes only an output stream as an argument; this uses the UTF-8 encoding by default.

You set indentation by calling the serializer's `setIndentation()` method with an integer argument specifying the number of spaces:

```
output.setIndentation(2);
```

You can write an entire XML document to the serializer destination by calling the serializer's `write()` method with the document as an argument:

```
output.write(doc);
```

The DomainWriter application inserts a comment atop the XML document instead of appending it at the end of a parent node's children. This requires another method of the parent node, `insertChild()`, which is called with two arguments—the element to add and the integer position of the insertion:

Click here to view code image

```
Builder builder = new Builder();
Document doc = builder.build(arguments[0]);
Comment timestamp = new Comment("File created " +
    new java.util.Date());
doc.insertChild(timestamp, 0);
```

The comment is placed at position 0 atop the document, moving the `domains` tag down one line but remaining below the XML declaration.

Listing 19.6 is the application's source code.

LISTING 19.6 The Full Text of `DomainWriter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import nu.xom.*;
 5:
 6: public class DomainWriter {
 7:     public static void main(String[] args) throws IOException
{
 8:         try {
 9:             // Create a tree from an XML document
10:             // specified as a command-line argument
11:             Builder builder = new Builder();
12:             File xmlFile = new File("feed2.rss");
13:             Document doc = builder.build(xmlFile);
14:
15:             // Create a comment with the current time and
date
16:             Comment timestamp = new Comment("File created "
17:                 + new java.util.Date());
18:
19:             // Add the comment above everything else in the
20:             // document
21:             doc.insertChild(timestamp, 0);
22:
23:             // Create a file output stream to a new file
24:             FileOutputStream f = new
FileOutputStream("feed3.rss");
25:
26:             // Using a serializer with indention set to 2
spaces,
27:             // write the XML document to the file
28:             Serializer output = new Serializer(f, "ISO-8859-
1");
29:             output.setIndent(2);
30:             output.write(doc);
31:         } catch (ParsingException pe) {
32:             System.out.println("Parsing error: " +
pe.getMessage());
33:             pe.printStackTrace();
34:             System.exit(-1);
35:         }
36:     }
37: }
```

The DomainWriter application reads the file `feed2.rss` as input and creates a new modified version called `feed3.rss`.

Evaluating XOM

The applications you've created cover the core features of the main XOM package and are representative of its straightforward approach to XML processing.

There also are smaller `nu.xom.canonical`, `nu.xom.converters`, `nu.xom.xinclude`, and `nu.xom.xslt` packages to support XInclude, Extensible Stylesheet Language Transformations (XSLT), canonical XML serialization, and conversions between the XOM model for XML and the one used by DOM and SAX.

Listing 19.7 is an application that works with XML from a dynamic source: RSS feeds of recently updated web content from the feed's producer. The RssFilter application searches the feed for specified text in headlines, producing a new XML document that contains only the matching items and shorter indentation. It also modifies the feed's title and adds an RSS 0.91 document type declaration if one is needed in an RSS 0.91 format feed.

Create the RssFilter application in the `com.java21days` package in NetBeans.

LISTING 19.7 The Full Text of `RssFilter.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import nu.xom.*;
 4:
 5: public class RssFilter {
 6:     public static void main(String[] arguments) {
 7:
 8:         if (arguments.length < 2) {
 9:             System.out.println("Usage: java RssFilter file
term");
10:             System.exit(-1);
11:         }
12:
13:         // Save the RSS location and search term
14:         String rssFile = arguments[0];
15:         String term = arguments[1];
16:
17:         try {
18:             // Fill a tree with an RSS file's XML data
19:             // The file can be local or something on the
```

```
20:             // Web accessible via a URL.
21:             Builder bob = new Builder();
22:             Document doc = bob.build(rssFile);
23:
24:             // Get the file's root element (<rss>)
25:             Element rss = doc.getRootElement();
26:
27:             // Get the element's version attribute
28:             Attribute rssVersion =
rss.getAttribute("version");
29:             String version = rssVersion.getValue();
30:
31:             // Add the DTD for RSS 0.91 feeds, if needed
32:             if ( (version.equals("0.91")) &
33:                 (doc.getDocType() == null) ) {
34:
35:                 DocType rssDtd = new DocType("rss",
36:                 "http://my.netscape.com/publish/formats/rss-
0.91.dtd");
37:                 doc.insertChild(rssDtd, 0);
38:             }
39:
40:             // Get the first (and only) <channel> element
41:             Element channel =
rss.getFirstChildElement("channel");
42:
43:             // Get its <title> element
44:             Element title =
channel.getFirstChildElement("title");
45:             Text titleText = (Text) title.getChild(0);
46:
47:             // Change the title to reflect the search term
48:             titleText.setValue(titleText.getValue() +
49:                 ": Search for " + term + " articles");
50:
51:             // Get all of the <item> elements and loop
through them
52:             Elements items =
channel.getChildElements("item");
53:             for (int i = 0; i < items.size(); i++) {
54:                 // Get an <item> element
55:                 Element item = items.get(i);
56:
57:                 // Look for a <title> element inside it
58:                 Element iTitle =
item.getFirstChildElement("title");
59:
60:                 // If found, look for its contents
61:                 if (iTitle != null) {
62:                     Text iTitleText = (Text)
iTitle.getChild(0);
```

```
63:
64:                         // If the search text is not found in the
item,
65:                         // delete it from the tree
66:                         if (iTitleText.toString().indexOf(term)
== -1) {
67:                             channel.removeChild(item);
68:                         }
69:                     }
70:                 }
71:
72:                 // Display the results with a serializer
73:                 Serializer output = new Serializer(System.out);
74:                 output.setIndent(2);
75:                 output.write(doc);
76:             } catch (Exception exc) {
77:                 System.out.println("Error: " + exc.getMessage());
78:                 exc.printStackTrace();
79:             }
80:         }
81: }
```

Run the application after setting the command-line arguments by selecting Run, Set Project Configuration, Customize. The first argument is the feed to check, and the second is the word to search for in its titles. One feed that can be used to test the application is http://feeds.sportsfilter.com/sportsfilter from the SportsFilter weblog. Check it for a word such as soccer, NFL, Yankees, or Cowboys.

Partial output of using RssFilter to look for "NFL" in SportsFilter's RSS feed is displayed in Figure 19.3.

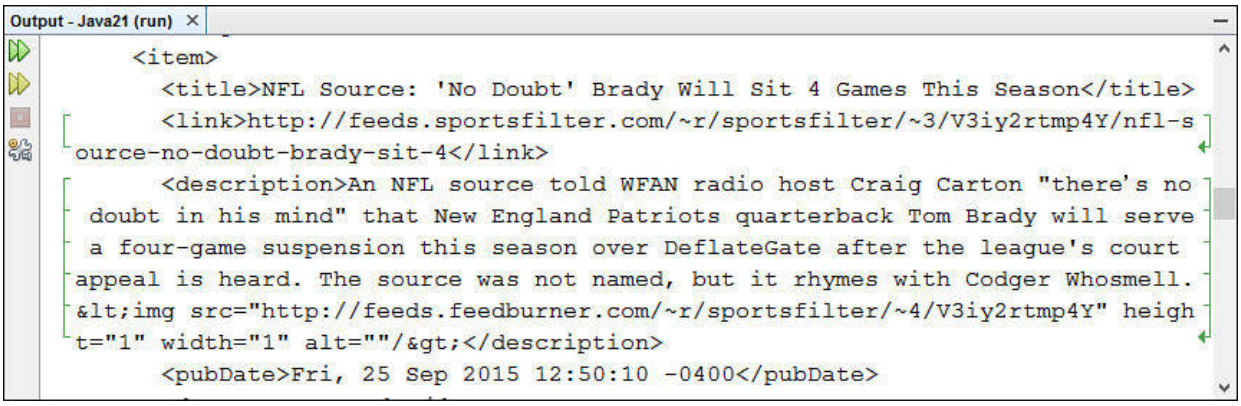


FIGURE 19.3 Reading XML data from a website's RSS feed.

Comments in the application's source code describe its functionality.

XOM's design is strongly informed by one overriding principle: enforced simplicity.

On the website for the class library, Elliotte Rusty Harold states that XOM "should help inexperienced developers do the right thing and keep them from doing the wrong thing. The learning curve needs to be really shallow, and that includes not relying on best practices that are known in the community but are not obvious at first glance."

The new class library is useful for Java programmers whose programs require a steady diet of XML.

Summary

Today you learned the basics of another popular format for data representation, Extensible Markup Language (XML), by exploring one of the most popular uses of XML—RSS feeds.

In some ways, XML is the data equivalent of the Java language. It liberates data from the software used to create it and the operating system the software runs on, just as Java can liberate software from a particular operating system.

By using a class library such as the open source XML Object Model (XOM) library, you can easily create and retrieve data from an XML file.

A big advantage of representing data using XML is that you can always get that data back. If you decide to move the data into a relational database or some other form, you can easily retrieve the information. The data being produced as RSS feeds can be mined by software in countless ways, today and in the future.

You also can transform XML into other forms such as HTML through a variety of technology, both in Java and through tools developed in other languages.

Q&A

Q What's the difference between RSS 1.0, RSS 2.0, and Atom?

A RSS 1.0 is a syndication format that employs the Resource Description Framework (RDF) to describe items in the feed. RSS 2.0 shares a common origin with RSS 1.0 but does not make use of RDF. Atom is another syndication format that was created after RSS 1.0 and RSS 2.0. The Internet Engineering Task Force (IETF) has adopted Atom as an Internet standard.

All three formats are suitable for offering web content in XML that can be read with a reader such as Feedly or My Yahoo! or that can be read by software and stored, manipulated, or transformed.

Q Why is Extensible Markup Language called XML instead of EML?

A None of the founders of the language appears to have documented the reason for choosing XML as the acronym. The general consensus in the XML community is that it was chosen because it "sounds cooler" than EML. Before you snicker at that explanation, the creator of the language you are learning chose the name Java for its programming language using the same criteria, turning down more technical-sounding alternatives such as DNA and WRL. (The name Ruby also was considered and rejected, but later was chosen for another language.)

It's possible that the founders of XML were trying to avoid confusion with a programming language called EML (Extended Machine Language), which predates XML.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What does RSS stand for?

   A. Really Simple Syndication

   B. RDF Site Summary

   C. Both

2. What method *cannot* be used to add text to an XML element using XOM?

   A. `addAttribute(String, String)`

   B. `appendChild(Text)`

   C. `appendChild(String)`

3. When all the opening element tags, closing element tags, and other markup are applied consistently in a document, what adjective describes the document?

   A. Validating

B. Parsable

C. Well-formed

Answers

1. C. One version, RSS 2.0, claims Really Simple Syndication as its name. The other, RSS 1.0, claims RDF Site Summary.
2. A. Answers B and C both work. One adds the contents of a `Text` element as the element's character data, and the other adds the string.
3. C. For data to be considered XML, it must be well-formed.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class NameDirectory {
    String[] names;
    int nameCount;

    public NameDirectory() {
        names = new String[20];
        nameCount = 0;
    }

    public void addName(String newName) {
        if (nameCount < 20) {
            // answer goes here
        }
    }
}
```

The `NameDirectory` class must be able to hold 20 different names. What statement should replace `// answer goes here` for the class to function correctly?

A. `names[nameCount] = newName;`

B. `names[nameCount] == newName;`

C. `names[nameCount++] = newName;`

D. `names[++nameCount] = newName;`

The answer is available on the book's website at www.java21days.com. Visit the Day 19 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Create a simple XML format to represent a book collection with three books and a Java application that searches for books with George R. R. Martin as the author, displaying any that it finds.
2. Create two applications: one that retrieves records from a database and produces an XML file that contains the same information, and a second application that reads data from that XML file and displays it.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 20. XML Web Services

Over the years, numerous attempts have been made to create a standard protocol for remote procedure calls (RPC). These are a way for one computer program to call a procedure in another program over a network such as the Internet.

Often, these protocols are completely language-agnostic. This allows a client program written in a language such as C++ to call a remote database server written in Java or something else without either side knowing (or caring) about its partner's implementation language.

Web services—networking programs that use the Web to offer data in a form easily digested by other software—are being employed to share account authentication between sites, facilitate e-commerce transactions between stores, provide business-to-business information exchange, and other innovative offerings.

A simple, useful example of this idea is XML-RPC, a protocol for using Hypertext Transfer Protocol (HTTP) and Extensible Markup Language (XML) for remote procedure calls. Today, you learn how to implement it in Java as the following topics are covered:

- How to communicate with another computer using XML-RPC
- How to structure an XML-RPC request and an XML-RPC response
- How to use XML-RPC in Java programs
- How to send an XML-RPC request
- How to receive an XML-RPC response

Introduction to XML-RPC

Java supports one well-established technique for remote procedure calling: remote method invocation (RMI).

RMI is designed to be a complex, robust solution to a large variety of remote computing tasks. This sophistication has been one of the hindrances to the adoption of RPC. The complexity required to implement some of these solutions can be more than a programmer wants to take on simply to exchange information over a network.

A much simpler alternative, XML-RPC, has become adopted for web services.

Client/server implementations of XML-RPC are available for most platforms and programming languages in widespread use.

XML-RPC exchanges information using a combination of HTTP, the protocol of the Web, and XML.

XML-RPC supports the following data types:

- `array`—A data structure that holds multiple elements of any of the other data types, including arrays
- `base64`—Binary data in Base 64 format
- `boolean`—True-false values that are either 1 (true) or 0 (false)
- `dateTime.iso8601`—A string containing the date and time in ISO 8601 format, such as 20150927T12:01:15 for 12:01 a.m. (and 15 seconds) on September 27, 2015
- `double`—8-byte signed floating-point numbers
- `int` (also called `i4`)—Signed integers ranging in value from –2,147,483,648 to 2,147,483,647, the same size as `int` values in Java
- `string`—Text
- `struct`—Name-value pairs of associated data where the name is a string and the value can be any of the other data types (comparable to the `HashMap` class in Java)

One thing noticeably absent from XML-RPC is a way to represent data as an object. The protocol wasn't designed with object-oriented programming in mind, but you can represent reasonably complex objects with the `array` and `struct` types.

By design, XML-RPC is a simple protocol for programming across a network. The protocol has been implemented on web services running on Windows, Macintosh, and Linux systems.

Note

The full XML-RPC specification is available on XML-RPC.com at www.xmlrpc.com/spec.

After the release of XML-RPC, the specification was extended to create another RPC protocol called Simple Object Access Protocol (SOAP).

SOAP shares design goals of XML-RPC and has been expanded to better support objects, user-defined data types, and other advanced features, resulting in a significantly more complex protocol. SOAP also has become widely popular for web services and other decentralized network programming.

Note

Because SOAP is an extension of XML-RPC, it raises the question of why the latter protocol is still in use. When SOAP came out and was considerably more complex than XML-RPC, some programmers chose to stick with the simpler protocol.

To find out more about SOAP and public servers that can be used with SOAP clients, visit the website www.xmethods.com.

Communicating with XML-RPC

XML-RPC is a protocol transmitted via HTTP, the standard for data exchange between web servers and web browsers. The information it transmits is not web pages. Instead, it is XML data encoded in a specific way.

Two kinds of data exchanges are conducted using XML-RPC: client requests and server responses.

Sending a Request

An XML-RPC request is XML data sent to a web server as part of an HTTP `post` request.

A `post` request normally is used to transmit data from a web browser to a web server. Java servlets, CGI programs, and other software collect the data from a `post` request and send back HTML in response. When you submit an email from a web page or vote in an online poll, you're using either `post` or a similar HTTP request called `get`.

XML-RPC simply uses HTTP as a convenient protocol for communicating with a server and receiving a response.

The request consists of two parts: the HTTP headers required by the `post` transmission, and the XML-RPC request, which is expressed as XML. Listing 20.1 is an example of an XML-RPC request.

LISTING 20.1 An XML-RPC Request

Click here to view code image

```
 1: POST /XMLRPC HTTP/1.0
 2: Host: www.advogato.org
 3: Connection: Close
 4: Content-Type: text/xml
 5: Content-Length: 151
 6: User-Agent: OSE/XML-RPC
 7:
 8: <?xml version="1.0"?>
 9: <methodCall>
10:    <methodName>test.square</methodName>
11:    <params>
12:       <param>
13:          <value>
14:             <int>13</int>
15:          </value>
16:       </param>
17:    </params>
18: </methodCall>
```

In Listing 20.1, lines 1–6 are the HTTP headers, and lines 8–18 are the XML-RPC request. This listing tells you the following:

- The XML-RPC server is at www.advogato.org/XMLRPC (lines 1 and 2).
- The remote method being called is `test.square` (line 10).
- The method is being called with one argument, an integer with a value of `13` (lines 12–16).

Unlike their counterparts in Java, method names in an XML-RPC request do not include parentheses. They consist of the name of an object followed by a period and the name of the method, or simply the name of the method, depending on the XML-RPC server.

Caution

XML-RPC, which has been implemented in numerous computer-programming languages, has differences in terminology from Java: Methods are called procedures, and method arguments are called parameters. The Java terms are used often during today's lesson when Java programming techniques are discussed.

Responding to a Request

An XML-RPC response is XML data that is sent back from a web server like any other HTTP response. Again, XML-RPC piggybacks on an established process—a web server sending data via HTTP to a web browser—and uses it in a new way.

The response also consists of HTTP headers and an XML-RPC response in XML format.

Listing 20.2 is an example of an XML-RPC response.

LISTING 20.2 An XML-RPC Response

Click here to view code image

```
HTTP/1.0 200 OK
Date: Sat, 27 Sep 2015 04:44:13 GMT
Server: Apache/2.2.3·(CentOS)
ETag: "PbT9cMgXsXnw52OqREFNAA=="
Content-MD5: PbT9cMgXsXnw52OqREFNAA==
Content-Length: 157
Connection: close
Content-Type: text/xml

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>
        <int>169</int>
      </value>
    </param>
  </params>
</methodResponse>
```

In Listing 20.2, lines 1–8 are the HTTP headers, and lines 10–19 are the XML-RPC response. You can learn the following things from this listing:

- The response is 157 bytes in size and is in XML format (lines 6 and 8).
- The value returned by the remote method is an integer that equals 169 (line 15).

An XML-RPC response contains only one argument, contrary to what you might expect from the `params` tag in line 12. If the remote method does not return a value—for example, it might be a Java method that returns `void`—an XML-RPC server still returns something.

This return value can be primitive data, strings, arrays of varying dimensions, and more sophisticated data structures such as key/value pairs (the kind of thing you could implement in Java using `HashMap`).

Note

The XML-RPC request and response examples were generated by a server run by the Advogato open source advocacy site. You can find out more about its XML-RPC server at www.advogato.org/xmlrpc.html.

Several XML-RPC debuggers on the Web can be used to call remote methods, which makes it much easier to determine if a client or server is working correctly. One is available at http://w3future.com/html/xmlrpcdebugger.html.

Choosing an XML-RPC Implementation

Although you can work with XML-RPC by creating your own classes to read and write XML and exchange data over the Internet, an easier route is to use a preexisting Java Class Library that supports XML-RPC.

One of the most popular is Apache XML-RPC, an open source project managed by the developers of the Apache web server, Tomcat Java servlet engine, and other popular open source software.

The Apache XML-RPC project, which consists of the `org.apache.xmlrpc` package and three related packages, contains classes that can be used to implement an XML-RPC client and server with a small amount of your own code.

The project has a home page at the web address http://xml.apache.org/xmlrpc. Today's projects employ release 3.1.3. To use this project, you must download and install it.

Apache XML-RPC can be downloaded as either a ZIP archive or TAR.GZ archive.

Caution

If you have trouble downloading Apache XML-RPC from the Apache website, you can get it from the book's website. Visit www.java21days.com, go to the page for Day 20 and click the "Apache XML-RPC Library version 3.1.3" link. This is a ZIP archive containing all the project's files. The software is open source and can be shared under the Apache License.

Download the library and extract the files on a folder on your computer. When that's done, follow these steps to add Apache XML-RPC to NetBeans:

1. Choose Tools, Libraries. The Ant Library Manager opens.
2. Click New Library. The New Library dialog appears.
3. Enter `Apache XML-RPC 3.1.3` as the Library Name, and click OK.
4. Back in the Library Manager, click Add JAR/Folder.
5. Browse to the folder where you extracted the archive, and open it.
6. Open the lib subfolder and choose all five JAR files it contains: `commons-logging-1.1.jar`, `ws-commons-util-1.0.2.jar`, `xmlrpc-client-3.1.3.jar`, `xmlrpc-common-3.1.3.jar`, and `xmlrpc-server-3.1.3.jar`. (To select multiple files, hold down the Shift key as you click each file.)
7. Click Add JAR/Folder.
8. Back in the Library Manager, click OK.

After you have added the library to NetBeans, you need to add it to the current project so that you can use the Apache XML-RPC classes in today's programs:

1. In the Projects pane, look for the folder named Libraries below the `.java` files for the classes you have created.

2. Right-click the Libraries folder and choose Add Library. The Add Library dialog appears.
3. Choose Apache XML-RPC 3.1.3, and click OK.

The five JAR files composing this class library appear under Libraries in the pane.

After the library is set up, an `import` statement makes it easy to refer to the classes in a package, as in this example:

```
import org.apache.xmlrpc.*;
```

This makes it possible to refer to the classes in the main package, `org.apache.xmlrpc`, without using the full package name. You'll work with this package in the next two sections.

Using an XML-RPC Web Service

An XML-RPC client is a program that connects to a server, calls a method on a program on that server, and stores the result.

Using Apache XML-RPC, the process is comparable to calling any other method in Java. You don't have to create an XML request, parse an XML response, or connect to the server using one of Java's networking classes.

In the `org.apache.xmlrpc.client` package, the `XmlRpcClient` class represents a client. The client is set up with the `XmlRpcClientConfigImpl` class, which holds the configuration settings for the client.

The server is set by calling the configuration object's `setServerURL(`*`URL`*`)` method with a `URL` object that contains the server's address and port number.

After configuration is complete, the client's `setConfig()` method is called with that configuration as the only argument.

The following statements create a client to an XML-RPC client on the host `cadenhead.org` at the port 4413:

Click here to view code image

```
XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
URL server = new URL("http://cadenhead.org:4413/");
config.setServerURL(server);
XmlRpcClient client = new XmlRpcClient();
client.setConfig(config);
```

If you are calling a remote method with any arguments, they should be stored in an `ArrayList` object, a data structure that holds objects of different classes.

Note

Array lists were covered on Day 8, "Data Structures." They are part of the `java.util` package.

To work with array lists, call the `ArrayList()` constructor with no arguments, and call its `add(`*`Object`*`)` method with each object that should be added to the list. Objects can be of any class and must be added to the list in the order in which they are called in the remote method.

The following data types can be arguments to a remote method:

- `byte[]` arrays for `base64` data
- `Boolean` objects for `boolean` values
- `Date` objects for `dateTime.iso8601` values
- `Double` objects for `double` values
- `Integer` objects for `int` values
- `String` objects for `string` values
- `HashMap` objects for `struct` values
- `ArrayList` objects for arrays

The `Date`, `HashMap`, and `ArrayList` classes are in the `java.util` package.

For example, if an XML-RPC server has a method that takes `String` and `Double` arguments, the following code creates an array list that holds each argument:

Click here to view code image

```
String code = "conical";
Double xValue = new Double(175);
ArrayList parameters = new ArrayList();
parameters.add(code);
parameters.add(xValue);
```

To call the remote method on the XML-RPC server, call the `XmlRpcClient` object's `execute()` method with two arguments:

- The name of the method
- The array list that holds the method's arguments

The name of the method should be specified without any parentheses or arguments. An XML-RPC server usually documents the methods that it makes available to the public.

The `execute()` method returns an `Object` that contains the response. This object should be cast to one of the data types sent to a method as arguments: `Boolean`, `byte[]`, `Date`, `Double`, `Integer`, `String`, `HashMap`, or `ArrayList`.

Like other networking methods in Java, `execute()` throws an `XmlRpcException` exception if the server reports an XML-RPC error.

Objects returned by the `execute()` method have the following data types: `Boolean` for `boolean` XML-RPC values, `byte[]` for `base64` data, `Date` for `dateTime.iso8601` data, `Double` for `double` values, `Integer` for `int` (or `i4`) values, `String` for strings, `HashMap` for `struct` values, and `ArrayList` for arrays.

To see all this in a working program, enter the code shown in Listing 20.3 into the NetBeans editor as the class `SiteClient` in the package `com.java21days`.

LISTING 20.3 The Full Text of `SiteClient.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.util.*;
 6: import org.apache.xmlrpc.*;
 7: import org.apache.xmlrpc.client.*;
 8:
 9: public class SiteClient {
10:     public static void main(String arguments[]) {
11:         SiteClient client = new SiteClient();
12:         try {
13:             HashMap<String, String> resp =
client.getRandomSite();
14:             // Report the results
15:             if (resp.size() > 0) {
```

```
16:                 System.out.println("URL: " + resp.get("url")
17:                     + "\nTitle: " + resp.get("title")
18:                     + "\nDescription: " +
resp.get("description"));
19:             }
20:         } catch (IOException ioe) {
21:             System.out.println("Exception: " +
ioe.getMessage());
22:         } catch (XmlRpcException xre) {
23:             System.out.println("Exception: " +
xre.getMessage());
24:         }
25:     }
26:
27:     public HashMap getRandomSite()
28:         throws IOException, XmlRpcException {
29:
30:             // Create the client
31:             XmlRpcClientConfigImpl config = new
32:                 XmlRpcClientConfigImpl();
33:             URL server = new URL("http://localhost:4413/");
34:             config.setServerURL(server);
35:             XmlRpcClient client = new XmlRpcClient();
36:             client.setConfig(config);
37:             // Create the parameters for the request
38:             ArrayList params = new ArrayList();
39:             // Send the request and get the response
40:             HashMap result = (HashMap) client.execute(
41:                 "dmoz.getRandomSite", params);
42:             return result;
43:     }
44: }
```

The SiteClient application connects to the XML-RPC server and calls the `dmoz.getRandomSite()` method on the server with no arguments. When it works, this method returns a `HashMap` that contains the site's URL, title, and description in strings with the keys "url", "title", and "description".

This class can be run, but it won't work because the XML-RPC server hasn't been implemented yet.

Note

These random sites are culled from the database of the Open Directory Project, a directory of more than five million sites at www.dmoz.org. The project's data is available for redistribution by others at no cost under the terms of the Open Directory License. For more information, visit www.dmoz.org/help/getdata.html.

Creating an XML-RPC Web Service

An XML-RPC server is a program that receives a request from a client, calls a method in response to that request, and returns the result. The server maintains a list of methods that it allows clients to call; these are different Java classes called handlers.

Apache XML-RPC handles all the XML and networking itself, enabling you to focus on the task you want a remote method to accomplish.

There are several ways to serve methods remotely. The simplest is to use the `WebServer` class in the `org.apache.xmlrpc.webserver` package, which represents a simple HTTP web server that responds to only XML-RPC requests.

This class has two constructors:

- `WebServer(`*`int`*`)` creates a web server listening on the specified port number.
- `WebServer(`*`int`*`, `*`InetAddress`*`)` creates a web server at the specified port and IP address. The second argument is an object of the `java.net.InetAddress` class.

Both constructors throw `IOException` exceptions if an input/output problem occurs with creating and starting the server.

The web server has an `XmlRpcServer` object associated with it that handles tasks related to the protocol. This class is in another package, `org.apache.xmlrpc.server`. Call the web server's `getXmlRpcServer()` method with no arguments to retrieve it.

The following statements create a web server on port 4413 and an object for its XML-RPC server:

Click here to view code image

```
WebServer server = new WebServer(4413);
XmlRpcServer xmlRpcServer = server.getXmlRpcServer();
```

The web server does not contain the remote methods that clients call via XML-RPC. These reside in handlers.

Handlers are set up by another class in the `org.apache.xmlrpc.server` package, `PropertyHandlerMapping`. This class contains configuration settings for an XML-RPC server, which can be set with a properties file or by calling its methods. It can be created with no arguments to the constructor:

Click here to view code image

```
PropertyHandlerMapping phm = new PropertyHandlerMapping();
```

To add a handler, call the mapping object's `addHandler(`*`String`*`, `*`Object`*`)` method with two arguments.

The first argument to `addHandler()` is a name to give the handler, which can be anything you choose. Naming an XML-RPC method is comparable to naming a variable. Clients will use this name when calling remote methods.

The SiteClient application created earlier today called the remote method `dmoz.getRandomSite()`. The first part of this call—the text preceding the period—refers to a handler given the name dmoz.

The second argument to `addHandler()` is a `Class` object for the handler's class.

These statements add a handler named dmoz to the XML-RPC server's property mapping and then set the server to use that configuration:

Click here to view code image

```
phm.addHandler("dmoz", DmozHandlerImpl.class);
xmlRpcServer.setHandlerMapping(phm);
```

The `DmozHandlerImpl` class is the one that implements the `getRandomSite()` method and any others that can be called remotely over XML-RPC. You'll create this class in a moment.

A class that handles remote method calls can be any Java class that contains `public` methods that return a value, as long as the methods take arguments that correspond with data types supported by Apache XML-RPC: `boolean`, `byte[]`, `Date`, `double`, `HashMap`, `int`, `String`, and `ArrayList`.

You can put existing Java classes to use as XML-RPC handlers without modification as long as they do not contain `public` methods that should not be called and each `public` method returns a suitable value.

Caution

The suitability of return values relates to the Apache XML-RPC implementation rather than XML-RPC itself. Other implementations of the protocol are likely to have some differences in the data types of the arguments they take in remote method calls and the values they return.

Using Apache XML-RPC, the web server allows any public method in the handler to be called, so you should use access control to keep prying clients out of methods that should remain off limits.

As the first step toward creating an XML-RPC service, the following code creates a simple web server that takes XML-RPC requests. In NetBeans, use Listing 20.4 to create the `DmozServer` application.

LISTING 20.4 The Full Text of `DmozServer.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import org.apache.xmlrpc.*;
 5: import org.apache.xmlrpc.server.*;
 6: import org.apache.xmlrpc.webserver.*;
 7:
 8: public class DmozServer {
 9:     public static void main(String[] arguments) {
10:          try {
11:              startServer();
12:          } catch (IOException ioe) {
13:              System.out.println("Server error: " +
14:                  ioe.getMessage());
15:          } catch (XmlRpcException xre) {
16:              System.out.println("XML-RPC error: " +
17:                  xre.getMessage());
18:          }
19:     }
20:
21:     public static void startServer() throws IOException,
```

```
22:          XmlRpcException {
23:
24:          // Create the server
25:          System.out.println("Starting Dmoz server ...");
26:          WebServer server = new WebServer(4413);
27:          XmlRpcServer xmlRpcServer =
server.getXmlRpcServer();
28:          PropertyHandlerMapping phm = new
PropertyHandlerMapping();
29:
30:          // Register the handler
31:          phm.addHandler("dmoz", DmozHandlerImpl.class);
32:          xmlRpcServer.setHandlerMapping(phm);
33:
34:          // Start the server
35:          server.start();
36:          System.out.println("Accepting requests ...");
37:      }
38: }
```

This class can't be compiled successfully until you have created the handler class `DmozHandlerImpl` and a `DmozHandler` interface that it implements.

The DmozServer application creates a web server at port 4413 and an associated XML-RPC server in lines 26–27.

Using the server's property mapping, a handler is added to the server: a `DmozHandlerImpl` object given the name `"dmoz"`. The server's `start()` method is called to begin listening for requests.

That's all the code required to implement a functional XML-RPC server. Most of the work is in the remote methods you want a client to call. They don't require any special techniques as long as they are public and return a suitable value.

To give you a complete example you can test and modify to suit your own needs, the `DmozHandler` interface and `DmozHandlerImpl` class are provided in the next two listings.

The `DmozHandler` interface defines the public methods that can be called remotely over XML-RPC. Create a new empty Java file of this class name, and fill it with Listing 20.5.

LISTING 20.5 The Full Text of `DmozHandler.java`

Click here to view code image

```
1: package com.java21days;
2:
3: import java.util.*;
4:
5: public interface DmozHandler {
6:     public HashMap getRandomSite();
7: }
```

This interface contains one method, `getRandomSite()`, which returns a `HashMap`. No other methods can be called.

The `DmozHandlerImpl` class is an implementation of the `DmozHandler` interface.

The techniques employed in this class were covered during Day 18, "Accessing Databases with JDBC 4.2 and Derby." They are a good review of how to use JDBC to retrieve records from a database—in this example, a MySQL database called `cool`.

Enter the code shown in Listing 20.6 in NetBeans as the class `DmozHandlerImpl`.

LISTING 20.6 The Full Text of `DmozHandlerImpl.java`

Click here to view code image

```
 1: package com.java21days;
 2:
 3: import java.sql.*;
 4: import java.util.*;
 5:
 6: public class DmozHandlerImpl implements  DmozHandler {
 7:
 8:     public HashMap getRandomSite() {
 9:         Connection conn = getMySqlConnection();
10:         HashMap<String, String> response = new HashMap<>();
11:         try {
12:             Statement st = conn.createStatement();
13:             ResultSet rec = st.executeQuery(
14:                 "SELECT * FROM cooldata ORDER BY RAND()
LIMIT 1");
15:             if (rec.next()) {
16:                 response.put("url", rec.getString("url"));
17:                 response.put("title",
```

```
rec.getString("title"));
18:                     response.put("description",
19:                         rec.getString("description"));
20:                 } else {
21:                     response.put("error", "no database record
found");
22:                 }
23:                 st.close();
24:                 rec.close();
25:                 conn.close();
26:             } catch (SQLException sqe) {
27:                 response.put("error", sqe.getMessage());
28:             }
29:             return response;
30:         }
31:
32:         private Connection getMySqlConnection() {
33:             Connection conn = null;
34:             String data = "jdbc:mysql://localhost/cool";
35:             try {
36:                 Class.forName("com.mysql.jdbc.Driver");
37:                 conn = DriverManager.getConnection(
38:                     data, "cool", "mrfreeze");
39:             } catch (SQLException s) {
40:                 System.out.println("SQL Error: " + s.toString()
+ " "
41:                     + s.getErrorCode() + " " + s.getSQLState());
42:             } catch (Exception e) {
43:                 System.out.println("Error: " + e.toString()
44:                     + e.getMessage());
45:             }
46:             return conn;
47:         }
48: }
```

Lines 34–38 of the DmozHandlerImpl application should be changed to reflect your own database, username, and password. In this class, a MySQL database named `cool` is accessed on the local computer with the username "cool" and the password "mrfreeze". You also might need to change the rest of the string used to connect to the database, depending on your driver.

When the server is up and running, you can run `SiteClient` to see the data from a randomly selected website, as shown in Figure 20.1.

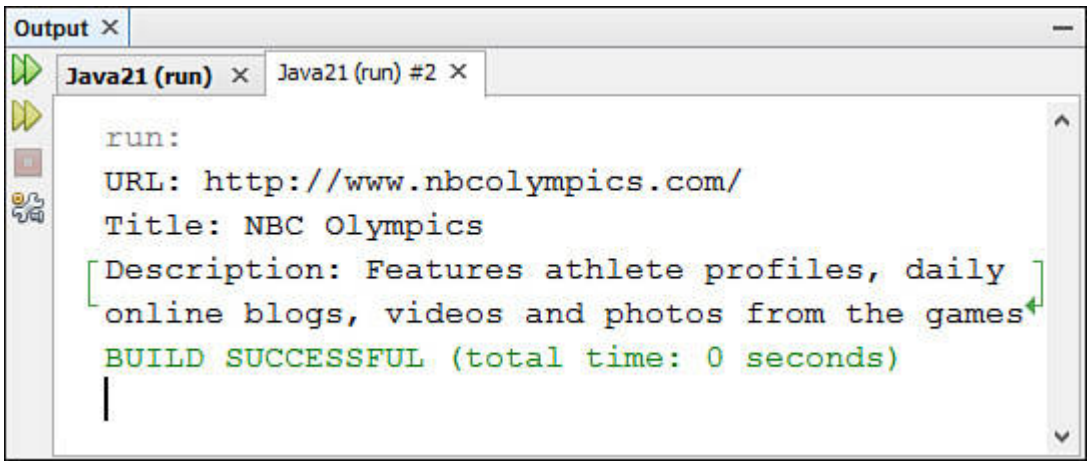


FIGURE 20.1 Receiving Dmoz website data over XML-RPC.

Note

Running this particular XML-RPC server also requires a database. To download a MySQL database containing information on 1,000 websites from the Open Directory Project, visit this book's website at www.java21days.com and open the Day 20 page. The database is in a file named `dmozdata.dat` and is a text file of SQL commands that can be used to create the database on a MySQL server.

Summary

XML-RPC has been described as the "lowest common denominator" of remote procedure call protocols, but this isn't considered an insult by its originators. Most attempts to facilitate software communication over a network have been sophisticated, scaring off developers who have simpler needs.

The XML-RPC protocol can be used to exchange information with any software that supports HTTP, the *lingua franca* of the Web, and XML, a highly popular, structured format for data.

By looking at XML-RPC requests and responses, you should be able to figure out how to use the protocol even without reading the protocol specification.

However, as implementations such as Apache XML-RPC become more extensive, you can begin using it quickly without ever looking at the protocol.

Q&A

Q When I try to return a `String` array from a remote method, Apache XML-RPC responds with an `XmlRpcException` that states that the object is not supported. Which objects does it support?

A Apache XML-RPC returns the following data types: `Boolean` for `boolean` XML-RPC values, `byte[]` for `base64` data, `Date` for `dateTime.iso8601` data, `Double` for `double` values, `Integer` for `int` (or `i4`) values, `String` for strings, `HashMap` for `struct` values, or `ArrayList` for arrays.

These are specific to Apache XML-RPC. Other class libraries that support this format may work with different data types and classes in Java. Consult the documentation for those libraries.

Q I'm writing an XML-RPC client to call a method that returns binary data (`base64`, in other words). The `execute()` method of `XmlRpcClient` returns an object instead of an array of bytes. How do I convert this?

A Arrays are objects in Java, so you can use casting to convert the object returned by `execute()` to an array of bytes (assuming that the object really is an array). The following statement accomplishes this on an object named `fromServer` that contains a byte array:

Click here to view code image

```
byte[] data = (byte[]) fromServer;
```

Quiz

Review today's material by taking this three-question quiz.

Questions

1. Which popular Internet protocol does XML-RPC not require?

A. HTML

B. HTTP

C. XML

2. Which XML-RPC data type would be best suited to hold the number 8.67?

   A. `boolean`

   B. `double`

   C. `int`

3. Which XML tag indicates that the data is an XML-RPC request?

   A. `methodCall`

   B. `methodResponse`

   C. `params`

Answers

1. A. XML-RPC uses HTTP (Hypertext Transfer Protocol) to transport data that is formatted as XML (Extensible Markup Language). HTML (Hypertext Markup Language) is not used.
2. B. All floating-point numbers such as 8.67 are represented by the `double` type in XML-RPC. There are not two different floating-point types, as there are in Java (`float` and `double`).
3. A. The `methodCall` tag is used only in requests, `methodResponse` is used only in responses, and `params` is used in both.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class Operation {
    public static void main(String[] arguments) {
        int x = 1;
        int y = 3;
        if ((x != 1) && (y++ == 3)) {
            y = y + 2;
        }
```

```
        }
    }
```

What is the final value of `y`?

A. 3

B. 4

C. 5

D. 6

The answer is available on the book's website at www.java21days.com. Visit the Day 20 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. The programming site Advogato offers an XML-RPC interface to read member diaries at http://www.advogato.org/xmlrpc.html. Write an application that reads a member's last 10 diary entries.
2. The XML-RPC interface for the weblog update service Weblogs.com is at www.weblogs.com/api.html. Write a client and server that can send and receive the `weblogUpdates.ping` method.

Exercise solutions are offered on the book's website at www.java21days.com.

Day 21. Writing Android Apps with Java

Once viewed primarily as a language for programs on a web page, Java has established itself as a powerful general-purpose programming language that can be run on desktop computers, Internet servers, tablets, appliances, and many other platforms. One platform in particular has in the past decade become an exciting and commercially lucrative area for new Java development: Android.

The Android operating system began on cell phones and quickly became the brains for a large number of other devices. All programs on Android are written in Java.

These programs, which are called apps, are developed on a free open source mobile OS that's enormously popular. Android does not require costly development tools, licensing fees, or approval by the OS developer. Anyone can create, distribute, and sell apps.

On this final day of the book, you learn about the history of Android, the things that have made it a success, and what it takes to develop programs for this OS. You learn how to create apps and run them on Android devices and emulators.

Today, these topics are covered:

- Why Android was created
- How to code your first app
- How to organize an app
- How to design the app's user interface
- How to deploy an app on emulators
- How to deploy an app on an Android phone

The History of Android

In 2007, Google launched Android in collaboration with several other tech companies and mobile phone manufacturers. The company hoped to establish a new mobile platform to challenge the dominance of the Apple iPhone and RIM BlackBerry in that space. Unlike those devices, Android was designed to be open, nonproprietary, and easy for third parties to participate in. Google, Intel, Nvidia, Samsung, Sprint Nextel, and 29 other companies formed the Open Handset Alliance to promote the new platform.

Google released at no cost the Android Software Development Kit (SDK), a set of tools for developing apps that run on the OS. The T-Mobile G1, which was released in 2008, was the first phone running Android to hit the market.

Once considered an also-ran in mobile computing, Android exploded in popularity within two years to rival the iPhone and has far surpassed it in market share today. All major phone carriers offer Android phones, and the market for tablets and e-book readers is growing rapidly. The research firm IDC estimates that 79 percent of all smartphones are running Android, compared to 16.4 percent using iPhone.

Before Android came along, mobile software development required expensive programming tools and private developer programs. The phone makers had the power to decide who could create apps for them and what apps could be sold to their users.

Because of Android's open-source, nonproprietary nature, anyone can create and distribute apps on the platform. There's a nominal cost to submit apps to Google's app marketplace, but everything else is free.

The central hub of Android programming is the Android Developer site at http://developer.android.com. The site offers extensive tutorials and reference material about writing software for the OS. This site provides documentation for every class in Android's Java Class Library, tutorials for beginners, and an online reference.

Writing Android apps requires an integrated development environment (IDE) that supports the Android SDK. Although NetBeans from Oracle can be used to create apps, Google encourages programmers to use another free IDE called Android Studio, which incorporates the SDK's functionality.

Android Studio can be used to write Android apps, test them in an emulator that acts like an Android device, and deploy them on the real thing.

The Java language has been used primarily to write software that runs on a desktop computer, web server, or web browser. Android puts the language everywhere. The apps you create can be deployed on phones, tablets, and other mobile devices, going with your users everywhere they go.

When James Gosling created Java while he worked at Sun Microsystems in the ’90s, the company wanted it to be a language for devices such as phones, smart cards, and appliances. Its slogan was “Write once, run everywhere.”

That lofty goal was set aside when the language rose to prominence—first as a way to put interactive programs on web pages and then as a general-purpose language for desktop computers and servers.

Thanks to Android, that goal has been met. One industry estimate is that the operating system is running on three billion devices around the world.

Let’s make that count three billion and one.

Writing an Android App

Android apps are Java programs that use an application framework, a set of classes and files that make the job easier as long as you follow all the rules. You’ve already used a framework, Swing, to create graphical user interfaces. The Android SDK provides a framework that lays down a set of rules for how apps must be structured to run properly on Android devices.

Before you can write apps, you must install and configure three things: the Android SDK, Android Studio, and Android Plug-in for Eclipse.

Android Studio will set up the SDK and Plug-in during installation. To get Android Studio, visit the website developer.android.com/sdk and click the Download button (or link) on this page. The software requires 1.1GB of disk space.

You will be asked what components to install along with Android Studio. Options include the Android SDK and Android Virtual Device, a simulator that acts like a phone or tablet running the mobile OS. Install both of these components and any others the installation wizard recommends.

Note

If you can't find a version of Android Studio for your operating system on the main download page, there's another download page for developers at http://tools.android.com/download/studio. Go to the Stable releases page and choose version 1.3.1 (the one used for this book). You will be presented with several versions for Windows, Mac OS, and Linux.

After the installation wizard downloads the Android SDK and other components, you are asked to choose a theme for the Android Studio user interface (Figure 21.1).

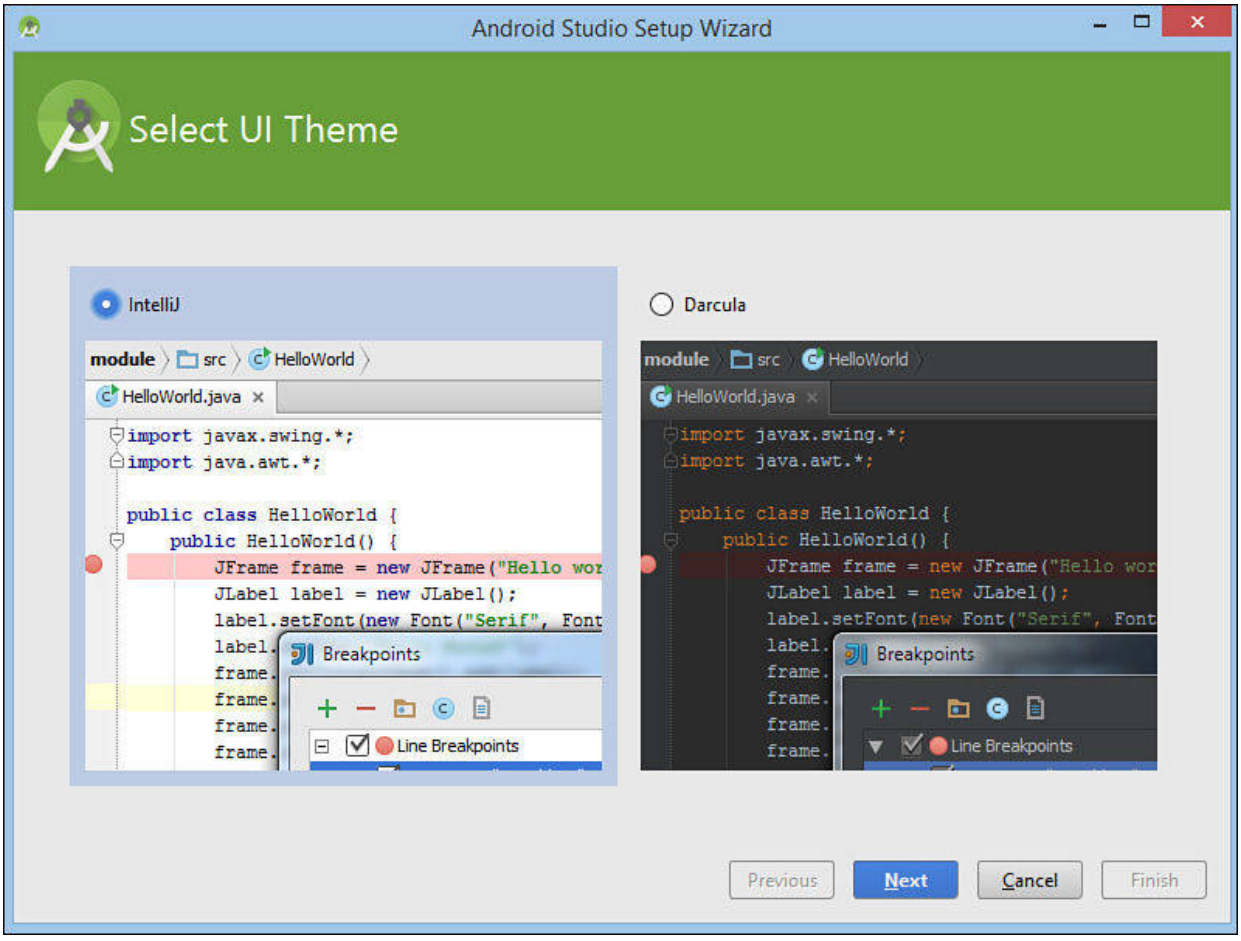


FIGURE 21.1 Choosing the user interface theme in Android Studio.

The IntelliJ theme resembles the user interface of NetBeans, so you may prefer it while you're getting started. This can be changed at any time.

As soon as you have Android Studio installed, you can get started.

Today's first project is Palindrome, an app that displays a line of text on an Android device.

The Android Studio Setup Wizard has a quick start menu that includes the comment Start a New Android Studio Project. Choose it, and the Create New Project wizard opens, as shown in Figure 21.2.

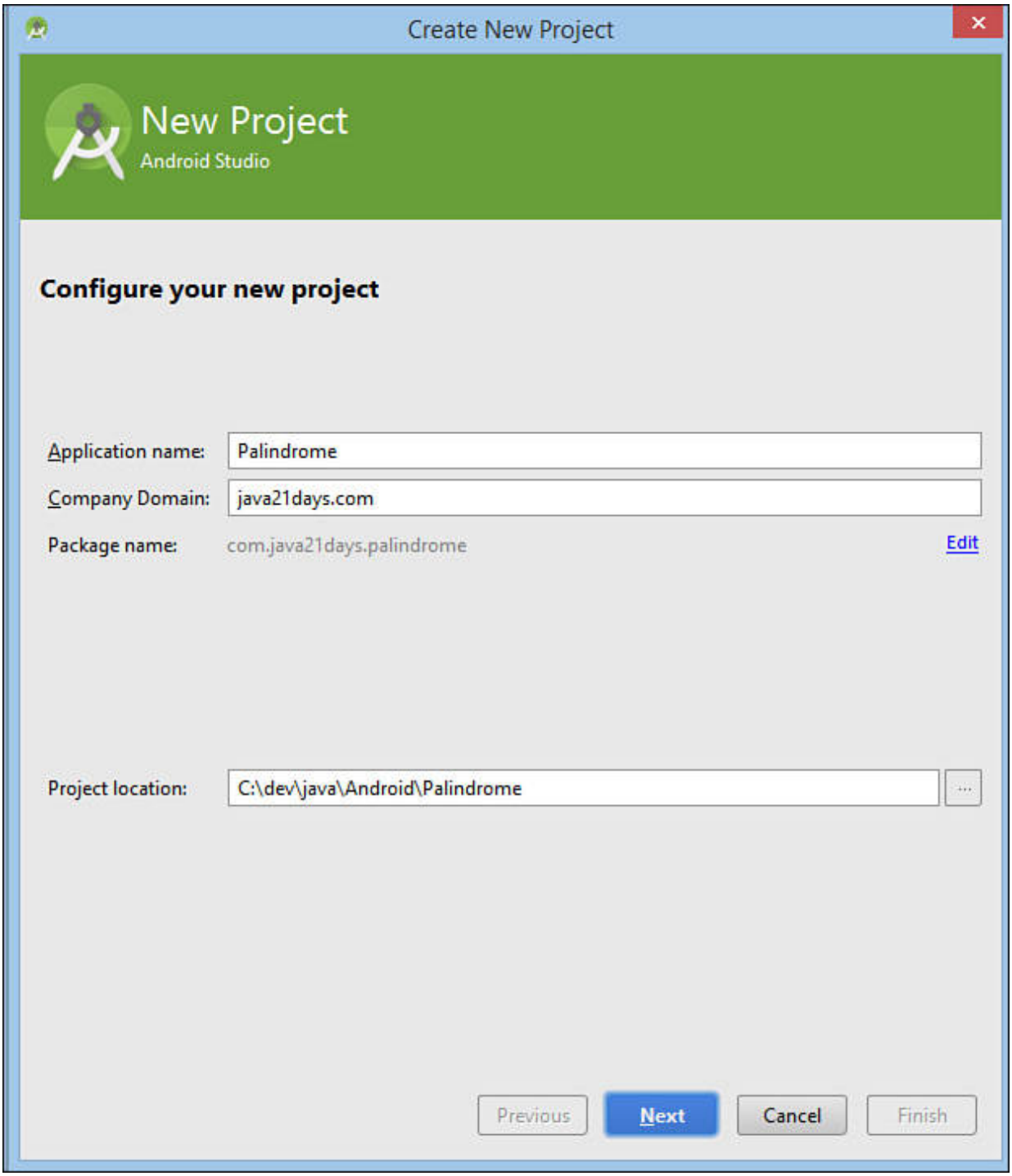


FIGURE 21.2 Starting a new Android project in Android Studio.

1. In the Application Name field, enter `Palindrome`.
2. In the Company Domain field, enter `java21days.com` or a domain name that you own. The Package Name field automatically reflects your

choice, forming a package name such as `com.java21days.palindrome`.

3. The Project Location field indicates the folder where project files will be stored. To change this, click the `...` button to the right of this field or enter a location.
4. Click Next. You'll be asked which type of Android devices to target with your app. The default is Phone and Tablet. Keep this default.
5. The Minimum SDK field determines the oldest version of Android OS that can run the app. The default is API 15: Android 4.0.3 (IceCreamSandwich). Keep this default and click Next.
6. You are asked whether to create an Activity, which is a class that performs a task. Choose Blank Activity; then click Next.
7. This activity must be customized. In the Activity Name field, enter `PalindromeActivity`. Accept all other defaults.
8. Click Finish.

Android Studio creates the new app and opens the new project in the IDE's user interface.

Note

Android Studio is a special version of IntelliJ IDEA, a commercial IDE for Java programmers that's now in its 14th major release. Though some versions of IntelliJ IDEA are sold commercially, Android Studio is free. If you like creating Java programs with this IDE, you can find out more about it at www.jetbrains.com/idea.

Organizing an Android Project

The Java programs you created in past days were made up primarily of class files. Sometimes a class needed the data in a file, such as a graphics file that contained a button icon or a text file read from an input stream.

Android projects always require external files. A new project is composed of about 20 files and folders, which are organized into a fixed folder structure. You can add files to those folders, but the starting files and folders must be present, or the app won't compile.

You can use the Android Studio Project pane, shown in Figure 21.3, to examine how a new Android project is organized. If the Project pane is not shown, click the Project tab along the left edge of the user interface to open it.

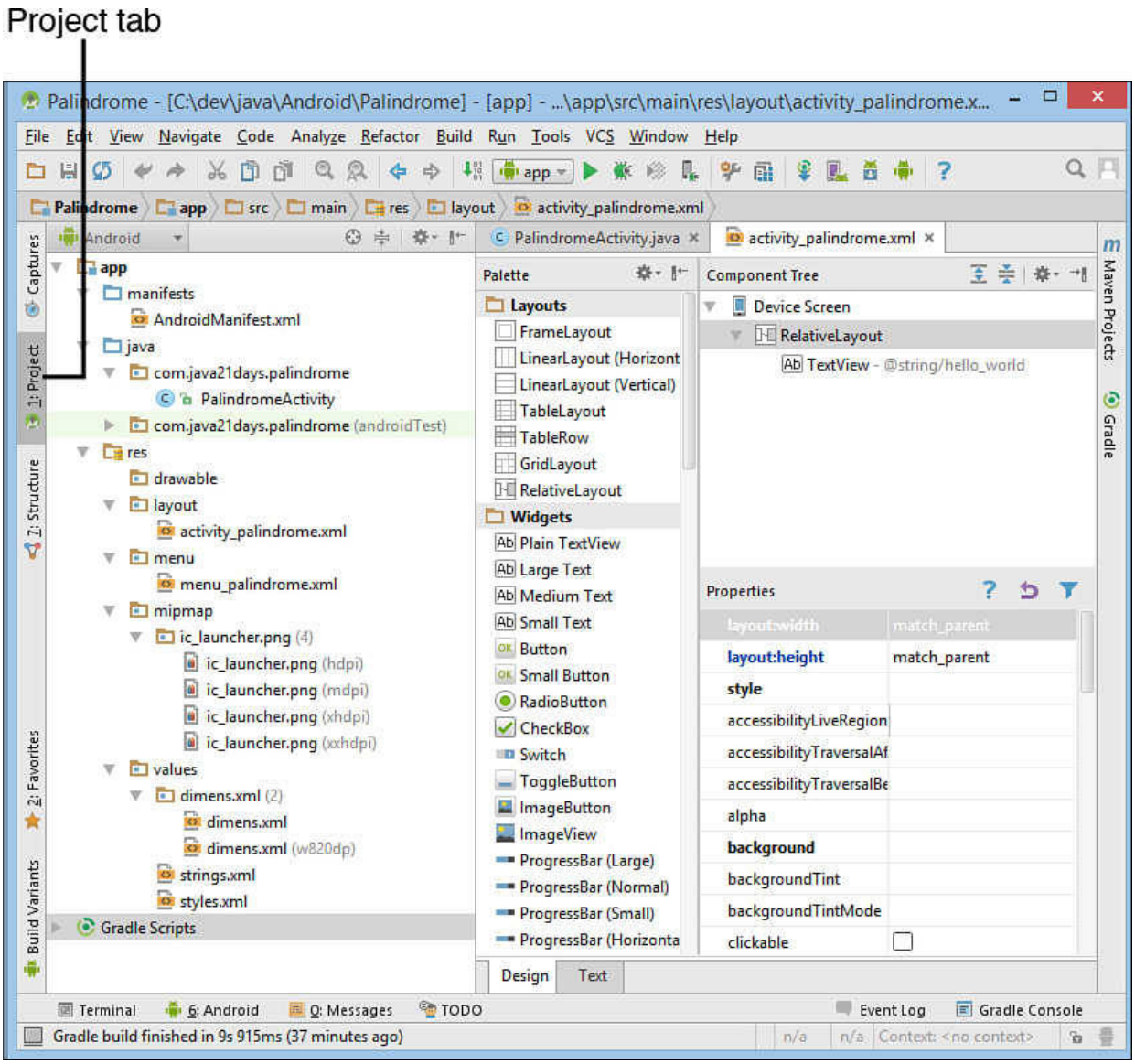


FIGURE 21.3 Viewing the components of an Android project.

In the Project pane, expand folders to familiarize yourself with the files and folders that a starting project contains. A new app such as Palindrome has these starting components:

- **`/java` folder**—The app's Java source code that you create.
- **`/java/com.java21days.palindrome/PalindromeActivity`**—The activity class created along with the project.

- **`/res`**—Application resources, which include animation, graphics, layout files, numbers, and strings. Subfolders `layout`, `values`, `drawable`, `menu`, and `midmap` hold specific resource types. These folders contain 10 resource files: four versions of `ic_launcher.png`, two versions of `dimens.xml`, `activity_palindrome.xml`, `menu_palindrome.xml`, `strings.xml`, and `styles.xml`.
- **`/manifests/AndroidManifest.xml`**—The app's configuration file.

These files and folders compose the app framework. The first thing an Android programmer must learn is what each of these components does and how they can be edited to create an app.

You can add files to the folders of the framework to create new functionality. For example, if an app has additional screens, they are added to the `/res/layout` folder.

Creating the Program

This framework can be run successfully as an app, but it wouldn't be much to look at, because nothing has been done to it yet.

The Palindrome app needs to be edited to display a palindrome, a sentence that reads the same forward and backward.

Java programs can display strings with the method `System.out.println(`*`String`*`)`, where literals and variables can be used as the argument. In Android apps, strings to be displayed are first saved in the resource file `strings.xml`, which is in the folder `/res/values`.

In the Project pane, find and expand this folder. Then double-click `strings.xml` to open it in the Resources editor. The file opens as an XML file. Above the contents of the file, a blue Open Editor link appears. Click this link to open strings.xml in the Translations editor, which is shown in Figure 21.4.

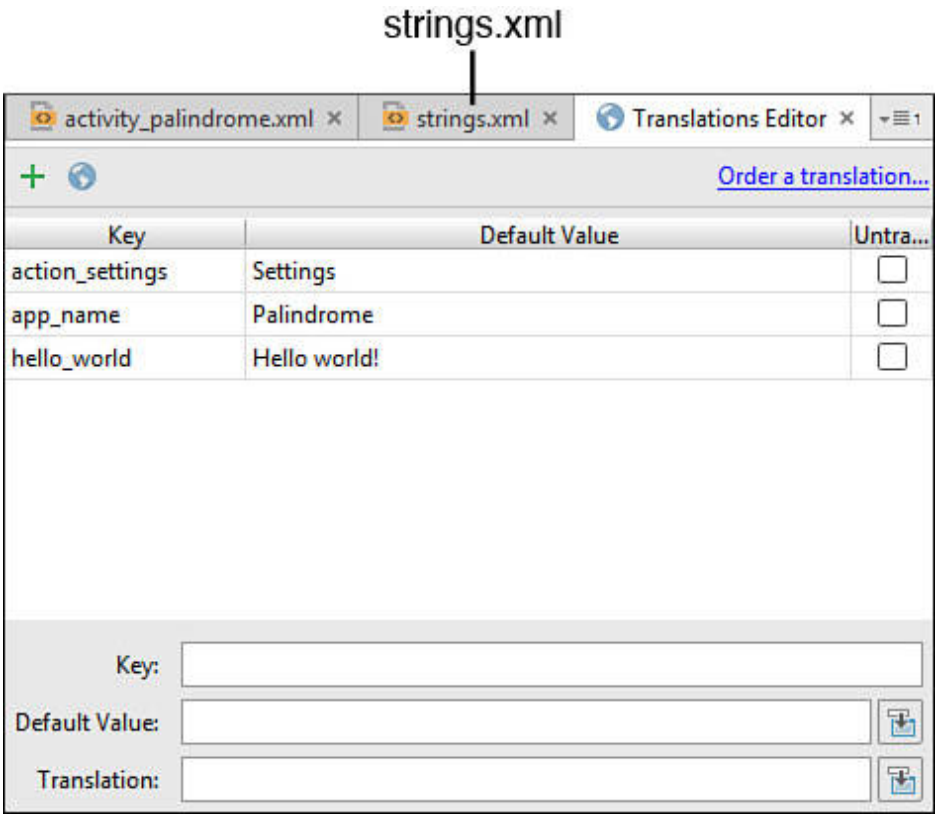


FIGURE 21.4 Creating an app's string resources.

Strings and other resources have a name and a value, comparable to Java variables. Three string resources appear in the editor with keys named `action_settings`, `app_name`, and `hello_world`. Resource keys must be lowercase, can contain no spaces, and can use only the underscore character (_) as punctuation.

To edit the value of one of these strings, click its table cell in the Default Value column of the editor.

In Figure 21.4, the `app_name` string resource defines the app's display name and was set by the Create New Project wizard. You can change it at any time by opening this resource and editing its value.

The `hello_world` string resource contains text that is displayed on the app's main screen, which at this point is its only screen. Change it from `Hello world!` to `Sit on a Potato Pan, Otis!`.

Android app resources such as `app_name` and `hello_world` are stored in XML files. The Resources editor is a simple XML editor. You also can directly edit the XML itself. Click the strings.xml tab at the top of the editor, shown in Figure 21.4, to load this file for direct editing.

The XML file `strings.xml` is open for editing in Figure 21.5.

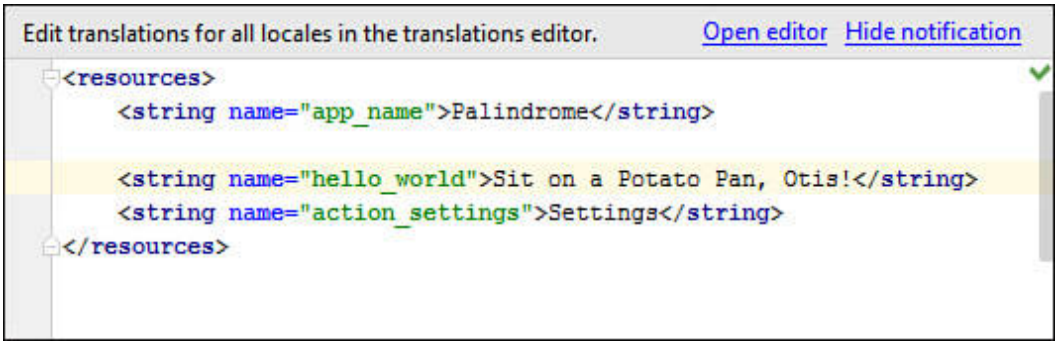


FIGURE 21.5 Editing an app's string resources as an XML file.

Everything in this XML file can be edited. You can give `app_name` a new value by replacing `Palindrome` with new text. You also can change the XML tags within the < and > characters. Each `string` element's `name` attribute defines the name of a string resource. The value is the character data contained within the opening and closing `string` tags.

Caution

Editing XML directly like this is much more error-prone than using the Translations editor. Making one typo in a tag causes the app to fail to compile. The only time you might want to edit XML is when Android Studio doesn't support something you need to define in a resource. This is never the case with strings, so use the Translations editor to create and modify them.

Click the Save All button in the Android Studio toolbar or choose File, Save All to save the change you made to `strings.xml`.

Running the App

To run your first app, choose the menu command Run, Run App. The Android emulator loads in its own window, which may take a minute or more, depending on how fast your computer is. The first thing that loads is a screen that displays an animated Android logo while the emulator continues to load the OS.

The emulator displays "Palindrome" in the app's title bar and one line of text on the app's screen, "Sit on a Potato Pan, Otis!," as shown in Figure 21.6. Controls to the right of the screen let you use the emulator like a phone with

the mouse. You also can click the screen, although in this app there's nothing to click.

FIGURE 21.6 Running an app in an emulator.

Click the Back button to close the Palindrome app and try out your new fake Android phone.

An emulator can simulate many things, such as connecting to the Internet over the computer's current connection and receiving phone calls and SMS messages.

If the emulator fails when you attempt to run the app, there's an error that occurs on some Windows and Mac OS computers. The following error message is displayed in Android Studio:

Output ▼

Click here to view code image

```
ERROR: x86 emulation currently requires hardware acceleration!
Please ensure Intel HAXM is properly installed and usable. CPU
acceleration status: HAX kernel module is not installed!
```

This error occurs on computers with an Intel processor when a program called the Intel Hardware Accelerated Execution Manager (HAXM) has not been installed. This program can be downloaded as part of the Android SDK. Before you go any further today, read Appendix C, "Fixing a Problem with the Android Studio Emulator."

Designing an Android App

Android apps can make use of SMS messaging, location-based services, touch screen input, and the rest of the device's functionality. For a final programming project, you'll create an app that can make a phone call, visit a website with the browser, and load a location using Google Maps.

After closing the Palindrome project with the menu command File, Close Project, create a new project in Android Studio by following these steps:

1. On the quick start menu, select Start a New Android Studio Project. The Create New Project wizard appears.
2. In the Application Name field, enter `Santa`.
3. Accept the other defaults and click Next. A Target Android Device pane appears.
4. Accept these defaults also and click Next. An activity pane appears.
5. Choose Blank Activity, and then click Next. You will be given a chance to customize the activity.
6. In the Activity Name field, enter `SantaActivity`.
7. Click Finish.

The project opens in the main Android Studio user interface.

Preparing Resources

Although Android apps are Java programs, a lot of the work required to create them is done in the Android Studio interface. You can accomplish many things in an Android app without writing Java code.

One thing you accomplish without programming is creating resources the app needs.

As shown earlier today, every new Android project begins with several folders and subfolders that contain resources. To examine these folders, expand the Project pane, and then expand `/res` and all its subfolders, as shown in Figure 21.7.

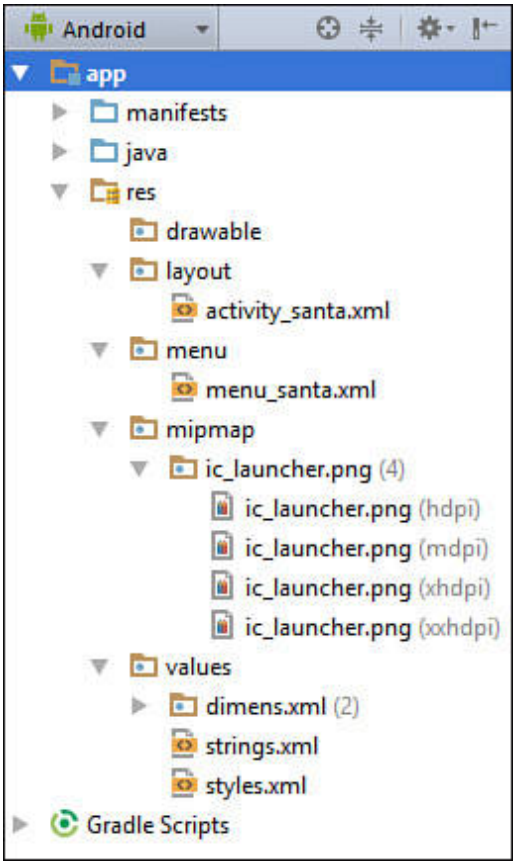


FIGURE 21.7 Examining an app's resources.

The project's starting resources are several graphics files, strings in `strings.xml`, and graphical user interface layout files that also are in XML format. Graphics must be in PNG, JPG, or GIF format. Two additional XML files that often are added to apps are `styles.xml`, which defines fonts, colors, and other visuals used in the app, and `dimens.xml`, which

holds dimensional measurements for text and other things that can be displayed in the app.

The `/res/mipmap` folder contains four versions of the file `ic_launcher.png`. This is the app's icon, the small graphic that represents it in application menus on an Android device. Each folder is for graphics of different screen resolutions.

The default icon for the app won't be used. A new graphics file, `santa.png`, will be brought into the project and designated as its icon in `AndroidManifest.xml`, the file that holds the app's configuration settings.

This book's website contains `santa.png` and four other graphics files needed by this app: `browser.png`, `maps.png`, `northpole.png`, and `phone.png`. Load www.java21days.com in your browser, open the site for this edition of the book, and click the Day 21 link. The graphics files are linked on this page. Download all five and save them to a folder on your computer (or the desktop).

Android's support for multiple resolutions is important for optimizing an app for different devices, but for this project it would be overkill.

You can add files to an Android project using drag and drop. Outside of Android Studio, open the folder where the five files for this app were downloaded from the book's website and copy them. Return to Android Studio, right-click drawable in the Project pane, and choose Paste.

Caution

Resource filenames can contain only lowercase letters, numbers, the underscore character _, and the period character (.).

Android apps identify resources using their filenames with the extension removed. This becomes the resource's ID, which is how it will be referred to in code. The files dragged into this project have the IDs `browser`, `maps`, `northpole`, `phone`, and `santa`. No two resources can have the same ID, except for versions of the same graphics file in the four `ic_launcher.png` folders, which are treated as a single resource. An app can't be compiled in Android Studio if two resources have the same ID.

Configuring a Manifest File

You can designate `santa.png` as the Santa app's icon by editing the resource file `AndroidManifest.xml`.

This file contains the app's configuration settings. Like `strings.xml`, this is an XML file that can be edited manually, but there's no special editor that can be used instead.

To choose the proper icon for the app, follow these steps:

1. In the Project pane, double-click `AndroidManifest.xml`. The XML file opens for editing, as shown in Figure 21.8.

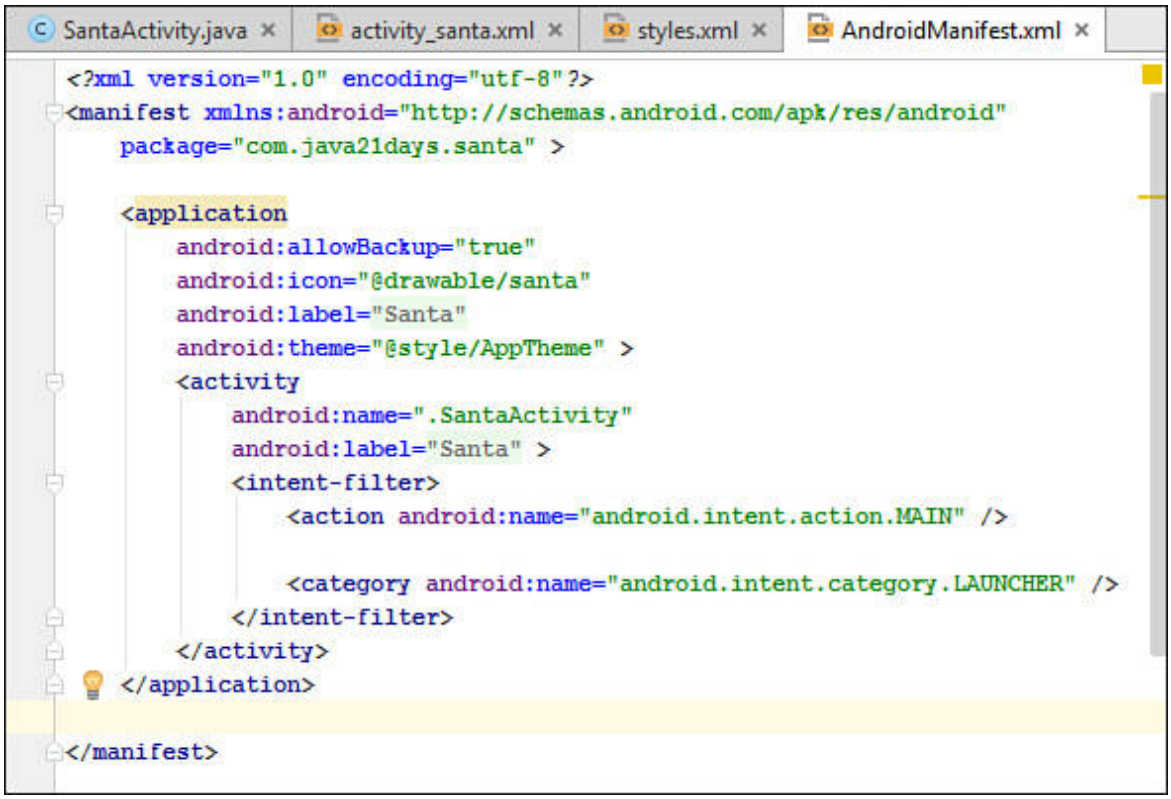


FIGURE 21.8 Editing an app's `AndroidManifest.xml` file.

2. Find the line where the attribute android:icon is set to the value "@mipmap/ic_launcher".
3. Change the value to "@drawable/santa".

The app's icon becomes a Santa icon.

Designing the Graphical User Interface

The graphical user interface for an Android app does not use Swing, because Android has its own library of user interface widgets. An app's graphical user interface is created as a collection of layouts, which are containers that hold text fields, buttons, graphics, and other widgets.

Each screen displayed to a user can have a single layout or multiple layouts. There are layouts that organize widgets into a table, stack them vertically or horizontally, and arrange them in other ways.

The Santa app has a single screen with buttons to contact Santa.

An app could be presented as multiple screens:

- A splash screen displays while the app loads.
- A menu screen contains buttons to access the other screens.
- A help screen explains how to use the app.
- A credits screen names the app's developer.
- A main screen accomplishes the app's purpose.

All of an app's screens are stored in the `/res/layout` folder. This new project was created with an `activity_santa.xml` file in this folder that's set up to display when the app loads.

To work on this screen layout, double-click `activity_santa.xml` in the Project pane. The screen opens for editing, as shown in Figure 21.9.

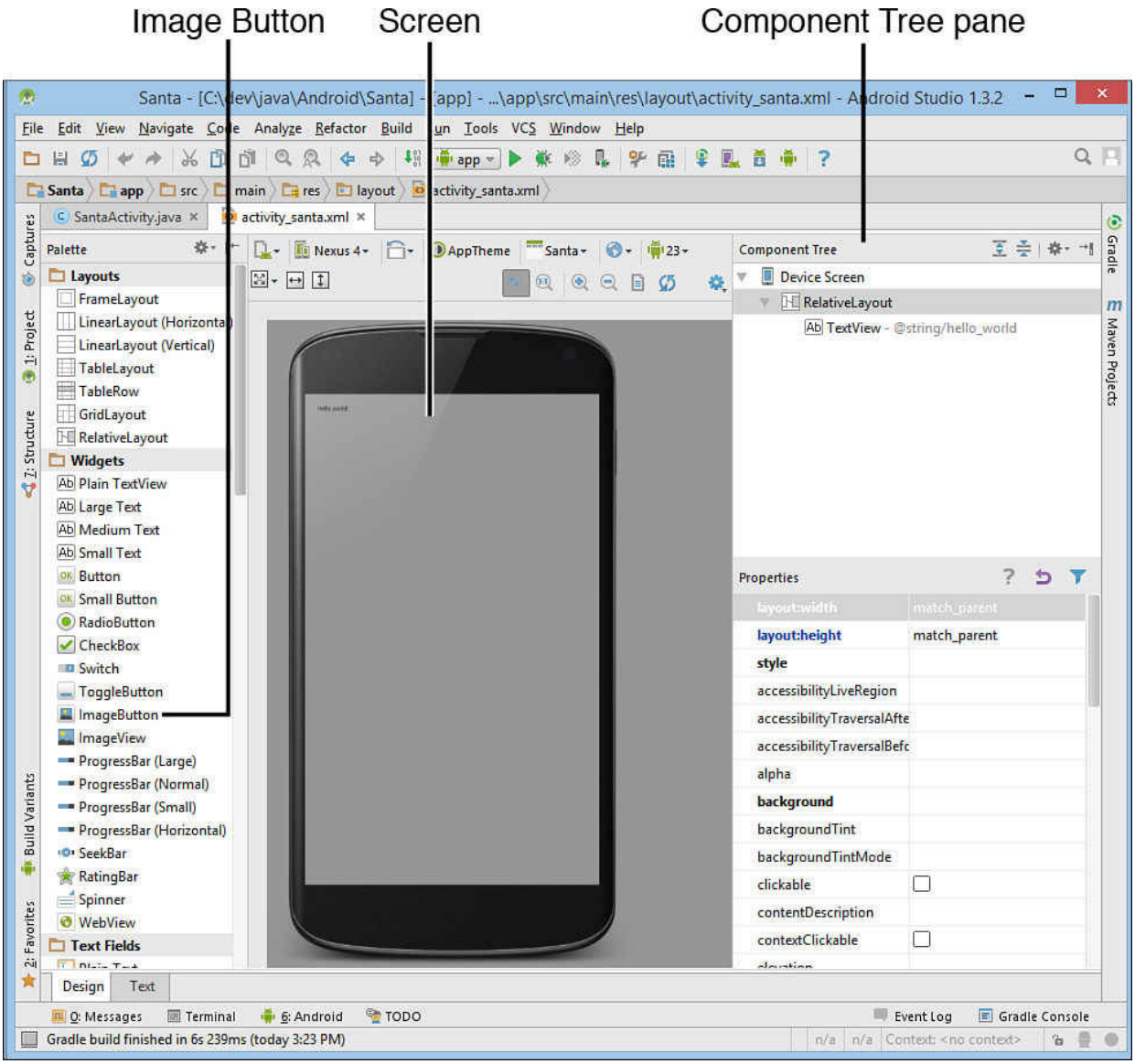


FIGURE 21.9 Editing an activity's graphical user interface.

Along the left side of the screen editor is a Palette pane with user interface widgets that can be dragged onto the screen.

Three graphical buttons, which are called `ImageButton` widgets in Android, must be added to the screen. Follow these steps:

1. On the app screen, click the widget that contains the text "Hello World"; a blue rectangle appears around the widget.
2. Press your keyboard's Delete key. The widget is removed.
3. Drag the layout widget LinearLayout (Horizontal) to the screen. This will cause components you place inside it to be arranged horizontally from left to right.

4. Drag an `ImageButton` widget (shown in Figure 21.9) from the Palette to the screen. A blue rectangle appears where the widget has been placed.
5. Double-click the rectangle. A dialog appears with the fields Src, which is empty, and ID, which is set to the value "imageButton".
6. Click the ... button next to the Src field. A Resources dialog appears.
7. Scroll down and choose the resource `phone`; then click OK. An image button with a Dialer icon appears.
8. Drag another `ImageButton` widget to the screen and place it immediately to the right of the Dialer button.
9. Use the same procedure in steps 4–6 to assign it the resource `browser`. Click OK. A Browser button appears.
10. Drag a third `ImageButton` to the screen to the right of the Browser button. Repeat steps 4–6 to assign it the resource `maps`. Click OK. A Maps icon appears. All three icons are lined up from left to right.
11. In the Component Tree pane to the right of the screen editor, click the LinearLayout item. The screen's properties appear in the Properties pane below the Component Tree pane.
12. In the Properties pane, click the value for Background; then click the ... button next to it. The Reference dialog opens.
13. Choose `northpole` and click OK. The screen's background becomes a photo of Santa and his sled.
14. Click the Dialer button. The Properties pane loads this widget's properties.
15. Scroll down to the `On Click` property. In its Value field, enter `processClicks` (capitalized as shown).
16. Do the same for the Browser button, setting its `On Click` property to `processClicks`.
17. Do the same for the Maps button.
18. Click the Save All button.

The finished screen is shown in Figure 21.10.

FIGURE 21.10 Viewing an app's user interface.

Placing the image buttons inside the linear layout widget and making them align properly takes practice. If your screen doesn't exactly match Figure 21.10, that's okay, as long as all three buttons can be seen.

You can start over, if desired, by clicking the LinearLayout item in the Component Tree and pressing the Delete key. All of the image buttons disappear along with the layout manager.

Writing Code

Without writing any Java code, you have completed most of this project. Android app development is much easier when you have learned how to exploit the features of Android Studio that require no programming.

Apps are organized into activities, which are the tasks an app can perform. Every Activity is a Java class. When you created the Santa app, one of the

options you specified was that an activity named `SantaActivity` should be created. This class runs when the app loads.

The source code for `SantaActivity.java` is in the `/java/com.java21days.santa` folder in the Project pane. Double-click this file to open it in the source code editor.

The class starts out with the code shown in Listing 21.1.

LISTING 21.1 The Starting Text of `SantaActivity.java`

Click here to view code image

```
 1: package com.java21days.santa;
 2:
 3: import android.support.v7.app.AppCompatActivity;
 4: import android.os.Bundle;
 5: import android.view.Menu;
 6: import android.view.MenuItem;
 7:
 8: public class SantaActivity extends AppCompatActivity {
 9:
10:     @Override
11:     protected void onCreate(Bundle savedInstanceState) {
12:         super.onCreate(savedInstanceState);
13:         setContentView(R.layout.activity_santa);
14:     }
15:
16:     @Override
17:     public boolean onCreateOptionsMenu(Menu menu) {
18:         // Inflate the menu; this adds items to the action
bar
19:         getMenuInflater().inflate(R.menu.menu_santa, menu);
20:         return true;
21:     }
22:
23:     @Override
24:     public boolean onOptionsItemSelected(MenuItem item) {
25:         // Handle action bar item clicks here. The action bar
will
26:         // automatically handle clicks on Home/Up button, so
long
27:         // as you specify a parent activity in
AndroidManifest.xml.
28:         int id = item.getItemId();
29:
30:         // noinspection SimplifiableIfStatement
31:         if (id == R.id.action_settings) {
32:             return true;
```

```
33:         }
34:
35:         return super.onOptionsItemSelected(item);
36:     }
37: }
```

If you can't see all the `import` statements in the editor, click the + character next to `import ....`

All activities are subclasses of `AppCompatActivity` in the `android.support.v7.app` package, which contains the behavior to display a screen, receive user input, and save user preferences.

The `onCreate()` method defined in lines 11–14 is called when the class loads. The first thing this method does is call the same method in its superclass.

Next, it calls `setContentView()` to select the layout to display on the screen. The method's argument is the instance variable `R.layout.activity_santa`, which refers to the file `activity_santa.xml` in `/res/layout`. The ID has the name `activity_santa` because each resource ID is its filename with the extension removed.

The R in `R.layout.main` refers to `R.java`, a class that Android Studio creates automatically.

While you were creating this app's screen, you set the `On Click` property for all three buttons to the value `processClicks`. This causes a method called `processClicks()` to be called when a user clicks those widgets.

That method must be implemented in `SantaActivity`. Below the last line of the `onCreate()` method on line 14, add these statements:

Click here to view code image

```
public void processClicks(View display) {
    Intent action = null;
    int id = display.getId();
}
```

When you enter this code, the Android Studio source code editor detects that the classes View and Intent have not been imported. A blue dialog appears, inviting you to hit Alt+Enter to import each class. Press Alt+Enter to do this.

The `processClicks()` method takes one argument, a `View` object from the `android.view` package. Views are visual displays in an app. This

particular `View` is the screen containing the Dialer, Browser, and Maps buttons.

The `View` object's `getId()` method returns the ID of the button that was clicked: `imageButton1`, `imageButton2`, or `imageButton3`.

This ID, which is saved in an integer variable named `id`, can be used in a `switch` conditional that takes action based on what the user clicks:

```
switch (id) {
    case (R.id.imageButton):
        // ...
        break;
    case (R.id.imageButton2):
        // ...
        break;
    case (R.id.imageButton3):
        // ...
        break;
    default:
        break;
}
```

The first statement in the `processClicks()` method declares a variable for an `Intent` object:

```
Intent action;
```

The `Intent` class in the `android.content` package is how one `Activity` tells another `Activity` to do something. An `Intent` also can be used to communicate with the device running the app.

Three intents are created in the `processClicks()` method, each in a `case` section of the `switch` conditional:

Click here to view code image

```
action = new Intent(Intent.ACTION_DIAL, Uri.parse(
    "tel:877-446-6723"));
action = new Intent(Intent.ACTION_VIEW, Uri.parse(
    "http://www.noradsanta.org"));
action = new Intent(Intent.ACTION_VIEW, Uri.parse(
    "geo:0,0?q=101 Saint Nicholas Dr., North Pole, AK"));
```

An `Intent()` constructor takes two arguments:

- The action to take, selected by a class variable
- The data associated with the action

The three `Intents` tell the Android device to set up an outgoing phone call to Santa's NORAD hotline at 877-446-6723, visit the website www.noradsanta.org, and load Google Maps with an address at the North Pole.

The `startActivity (Intent)` statement turns an intent into action:

```
startActivity(action);
```

For security reasons, the call is not made by the first intent. Instead, the device's dialer is opened with that number ready to be called.

Listing 21.2 contains the full text of the `SantaActivity` class. Double-check the listing to be sure your code matches this listing.

LISTING 21.2 The Full Text of `SantaActivity.java`

Click here to view code image

```
 1: package com.java21days.santa;
 2:
 3: import android.content.Intent;
 4: import android.net.Uri;
 5: import android.support.v7.app.AppCompatActivity;
 6: import android.os.Bundle;
 7: import android.view.Menu;
 8: import android.view.MenuItem;
 9: import android.view.View;
10:
11: public class SantaActivity extends AppCompatActivity {
12:
13:     @Override
14:     protected void onCreate(Bundle savedInstanceState) {
15:         super.onCreate(savedInstanceState);
16:         setContentView(R.layout.activity_santa);
17:     }
18:
19:     public void processClicks(View display) {
20:         Intent action = null;
21:         int id = display.getId();
22:
23:         switch (id) {
24:             case (R.id.imageButton):
25:                 action = new Intent(Intent.ACTION_DIAL,
26:                     Uri.parse("tel:877-446-6723"));
27:                 break;
28:             case (R.id.imageButton2):
29:                 action = new Intent(Intent.ACTION_VIEW,
```

```
30:                     Uri.parse("http://www.noradsanta.org"));
31:                 break;
32:             case (R.id.imageButton3):
33:                 action = new Intent(Intent.ACTION_VIEW,
34:                     Uri.parse("geo:0,0?q=101 Saint Nicholas
Dr., North Pole, AK"));
35:                     break;
36:             default:
37:                 break;
38:             }
39:         startActivity(action);
40:     }
41:
42:     @Override
43:     public boolean onCreateOptionsMenu(Menu menu) {
44:         // Inflate the menu; this adds items to the action
bar
45:         getMenuInflater().inflate(R.menu.menu_santa, menu);
46:         return true;
47:     }
48:
49:     @Override
50:     public boolean onOptionsItemSelected(MenuItem item) {
51:         // Handle action bar item clicks here. The action bar
will
52:         // automatically handle clicks on Home/Up button, so
long
53:         // as you specify a parent activity in
AndroidManifest.xml.
54:         int id = item.getItemId();
55:
56:         //noinspection SimplifiableIfStatement
57:         if (id == R.id.action_settings) {
58:             return true;
59:         }
60:
61:         return super.onOptionsItemSelected(item);
62:     }
63: }
```

As you save the file, Android Studio automatically compiles the class if there are no errors. Otherwise, red Xs appear in the Project pane in any file where an error has been detected.

To run the app, choose the menu command Run, Run App. The emulator loads Android OS and then runs the Santa app.

In the emulator, the app's Dialer, Browser, and Map buttons all should work.

The app running in an emulator will look like it did when you were creating it in Android Studio.

Click the Back button to exit the app. If you look at the list of applications on your fake phone, it now includes a Santa icon (Figure 21.11). Click it to run the app again.

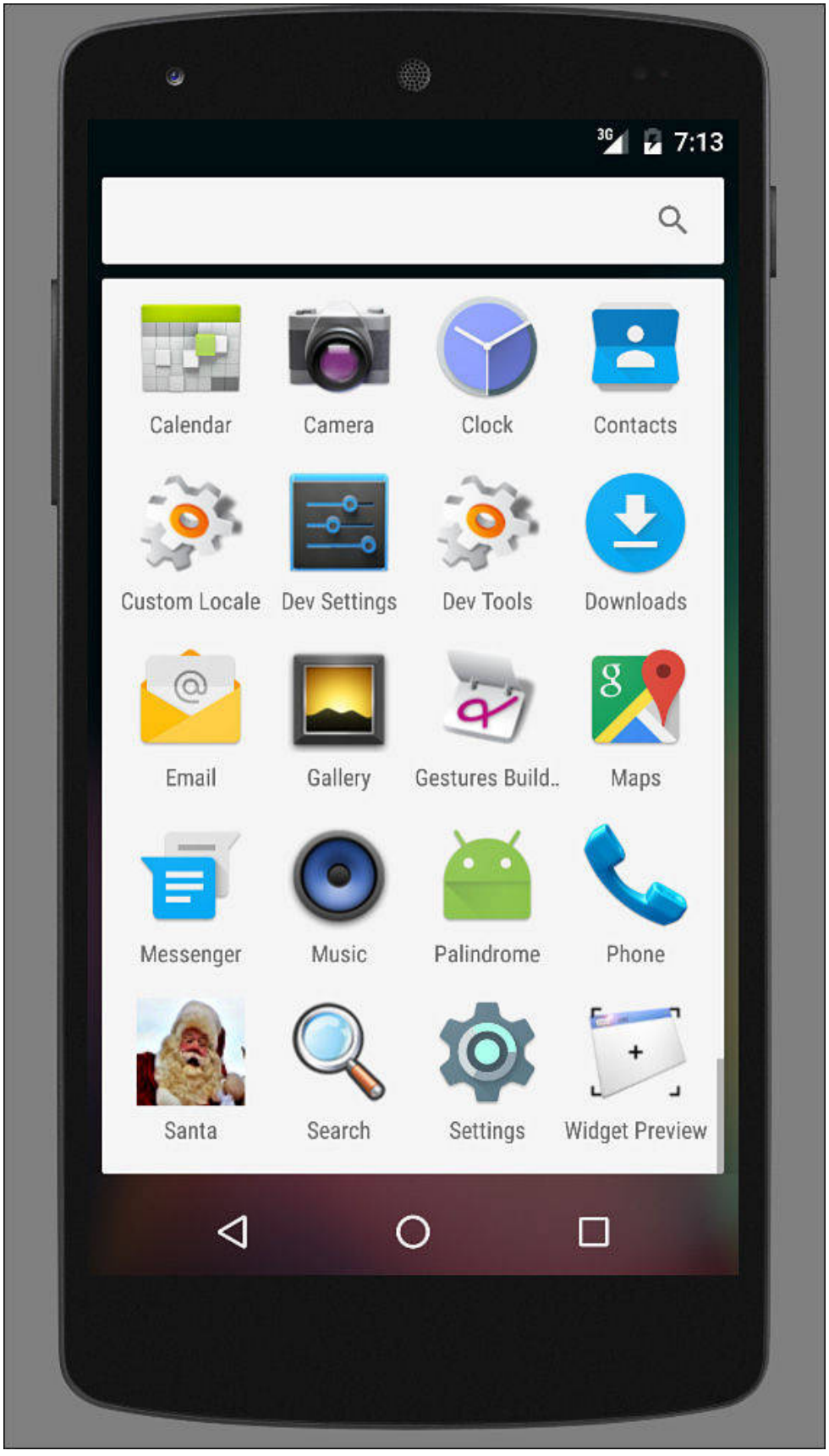

3G
7:13
Calendar
Camera
Clock
Contacts
Custom Locale
Dev Settings
Dev Tools
Downloads
Email
Gallery
Gestures Build..
Maps
Messenger
Music
Palindrome
Phone
Santa
Search
Settings
Widget Preview

FIGURE 21.11 Calling Santa Claus.

Note

For more on Android, refer to *Sams Teach Yourself Android Application Development in 24 Hours*, Fourth Edition, by Carmen Delessio, Lauren Darcey and Shane Conder (Sams, 2011, ISBN 0-672-33739-8). The Android Developer site also has tutorials and reference material at http://developer.android.com.

The apps you create and test in Android Studio also can be run on your Android phone. The process to make this happen varies, depending on the phone's manufacturer and its Android version.

To run apps currently under development, the phone must be set to developer mode and have USB debugging turned on. Open the phone's Settings app and look for the setting titled Developer Options (or something similar).

Turn on Developer Options and turn on USB debugging, one of several options that can be chosen when the phone is in developer mode. Plug in your phone's USB cord and connect it to the PC.

In Android Studio, choose Run, Run App. If the phone is set up properly, the Choose Device dialog now has a new option, as you can see in Figure 21.12.

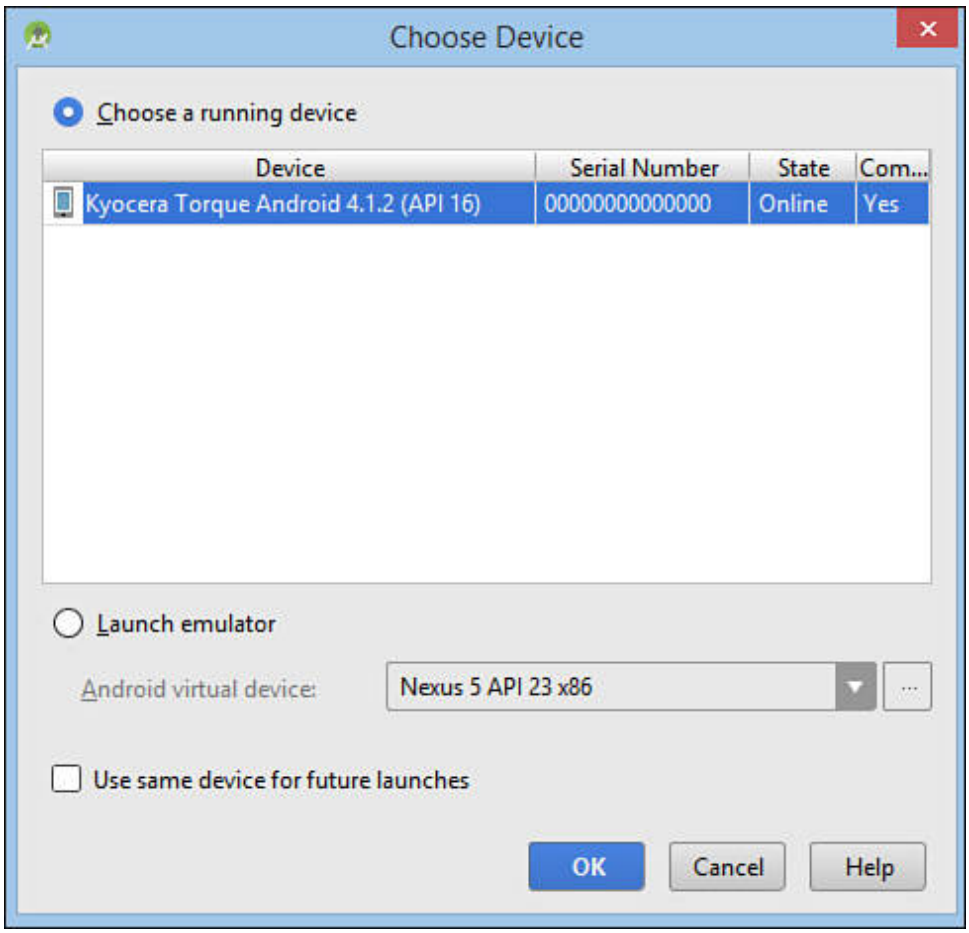


FIGURE 21.12 Running your app on a real Android phone.

Your phone will be listed in Choose Device, along with its name, Android version, and API version. Select the option Choose a Running Device, select your phone, and click OK.

The app will load in the phone and can be run like any other app. This is a permanent change, so even after you disconnect the USB cord, the app will still be there and you can run it by clicking its icon.

If your phone doesn't show up in the Choose Device dialog, the most likely cause is that your computer doesn't have the right USB driver.

There's a help page on the Android Developer site that documents how to find and install the proper driver. Visit http://developer.android.com/tools/extras/index.html. There are links for each of the manufacturers of Android phones that go to their driver download pages.

Follow the manufacturer's instructions for how to install the driver, making sure to uninstall an older driver version if you're asked to do that first.

Summary

For the past three weeks, you've had a chance to work with the syntax and the core classes that make up the Java language and the Java Class Library. You've ventured into sophisticated topics such as JDBC, Internet networking, and data structures, and you've explored class libraries such as the XML Object Model library and Android.

Now you are ready to tackle the biggest challenge yet: Turning an empty source code file into a robust and reliable program implemented as a set of Java classes using object-oriented programming.

This book has an official website at www.java21days.com with answers to frequently asked questions, source code for the entire book, error corrections, and supplementary material.

Now get to work on the next billion-dollar tech startup. In your IPO, don't forget the author who was there when you taught yourself Java.

Q&A

Q Do I need to create Android virtual devices for older versions of the Android SDK in Android Studio?

A Probably, because you want an app to run on as many versions of the Android operating system as possible. A large variety of Android devices are in use today, and not all are being updated to the current OS. Some can't be updated.

To ensure that your app has the widest possible audience, use the Android SDK Manager—available as one of the buttons in the Android Studio toolbar—to install older versions of Android.

More than 94 percent of devices are running Android version 2.3 (Gingerbread) or later, and more than 99 percent are running version 2.2 (Froyo) or later. Writing an app that works in early versions of Android will restrict the features it can use, because new Android capabilities that came out in subsequent releases won't be available.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What Android object enables an app to communicate with the device running the app?

   A. `Intent`

   B. `View`

   C. `Activity`

2. Which resource file contains an app's string resources?

   A. `main.xml`

   B. `strings.xml`

   C. `R.java`

3. Can an app have resource files named `icon.gif` and `icon.png`?

   A. Yes

   B. No

   C. Ask again later.

Answers

1. A. The `Intent` also can be used for one `Activity` to tell another to take an action.
2. B. The `main.xml` file is a screen, and `R.java` defines resource IDs.
3. B. No, because both files would be given the same identifier, `icon`.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

Given:

Click here to view code image

```
public class CharCase {
    public static void main(String[] arguments) {
        float x = 9;
        float y = 5;
        char c = '1';
        switch (c) {
            case 1:
```

```
                x = x + 2;
            case 2:
                x = x + 3;
            default:
                x = x + 1;
        }
        System.out.println("Value of x: " + x);
    }
}
```

What will be the value of `x` when it is displayed?

A. 9.0

B. 10.0

C. 11.0

D. The program will not compile.

The answer is available on the book's website at www.java21days.com. Visit the Day 21 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Modify the Palindrome app to display a different palindrome and show a graphic as the screen's background.
2. Modify the Santa app with the phone number, website address, and map location of another famous person.

Exercise solutions are offered on the book's website at www.java21days.com.

Week IV: Appendices

A Using the NetBeans Integrated Development Environment

B This Book's Web Site

C Fixing a Problem with the Android Studio Emulator

D Using the Java Development Kit

E Programming with the Java Development Kit

Appendix A. Using the NetBeans Integrated Development Environment

Although it's possible to create Java programs with nothing more than the Java Development Kit and a text editor, the experience is considerably more pleasant when you use an integrated development environment (IDE).

The first 20 days of this book employ NetBeans, a free IDE that Oracle offers to Java programmers. NetBeans is a program that makes it easier to organize, write, compile, and test Java software. It includes a project and file manager, graphical user interface designer, and many other tools. One killer feature is a code editor that automatically detects Java syntax errors as you type.

Now in version 8.0.2, NetBeans has become a favorite of professional Java developers, offering functionality and performance that used to be available only in commercial development tools at no cost. It's also one of the easiest IDEs for Java novices to use.

In this appendix, you install NetBeans and learn how to use it in projects created in this book.

Installing NetBeans

From inauspicious beginnings, the NetBeans IDE has grown to become one of the leading programming tools for Java developers. James Gosling, creator of the Java language, wrote in the Foreword to *NetBeans Field Guide*: "I use NetBeans for all my Java development." I've become a convert as well.

NetBeans supports all facets of Java programming for the three editions of the language—Java Standard Edition (JSE), Java Enterprise Edition (JEE), and Java Micro Edition (JME). It also supports web application development, web services, JavaBeans, and Android development.

You can download NetBeans for Windows, Mac OS, and Linux, from www.netbeans.org. NetBeans is available for download bundled with the Java Development Kit, but it's easy to install them separately.

If you'd like to ensure that you're downloading the same version of NetBeans used to write this book, visit the book's website at www.java21days.com. Click the book's cover to open the site for this edition, and then look for the

Download JDK 8 and Download NetBeans 8.0.2 links. You'll be steered to the proper files.

Tip

After you have installed NetBeans, you can use the IDE to get the latest version of the software. Choose the menu command Help, Check for Updates. (On Windows, you might need to run NetBeans as an administrator. To do this, right-click the NetBeans icon in a folder and choose Run as Administrator.)

Creating a New Project

The JDK and NetBeans are downloaded as installation wizards that set up the software on your system. You can install the software in any folder and menu group you like, but it's best to stick with the default setup options unless you have a good reason to do otherwise.

When you run NetBeans for the first time after installation, you see a start page that displays links to news, programming tutorials, and blogs, as shown in Figure A.1. You can read these within the IDE using NetBeans' built-in web browser.

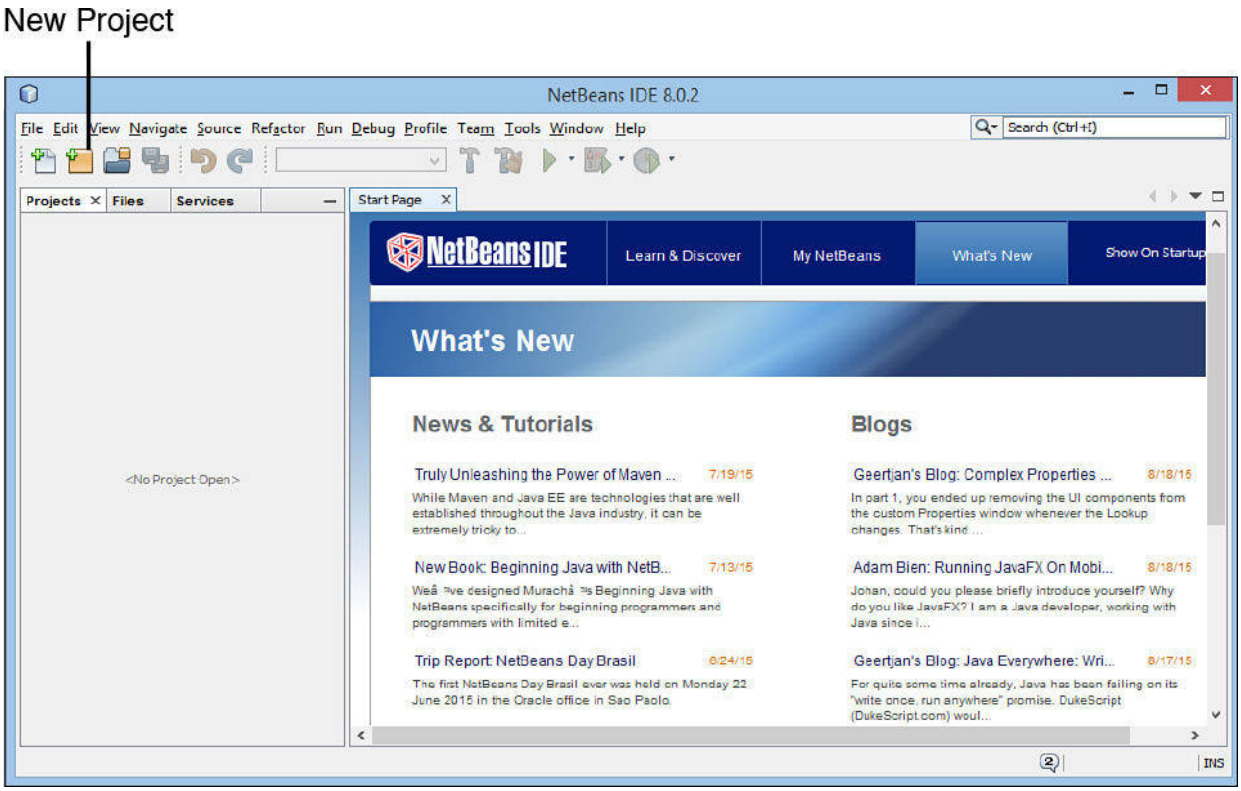


FIGURE A.1 The NetBeans user interface.

A NetBeans project consists of a set of related Java classes, files used by those classes, and Java class libraries. Each project has its own folder. You can explore and modify the files in the folder outside of NetBeans using text editors and other programming tools, like any other Java source code you create outside of NetBeans.

To begin a new project, click the New Project button shown in Figure A.1 or select File, New Project. The New Project Wizard opens, as shown in Figure A.2.

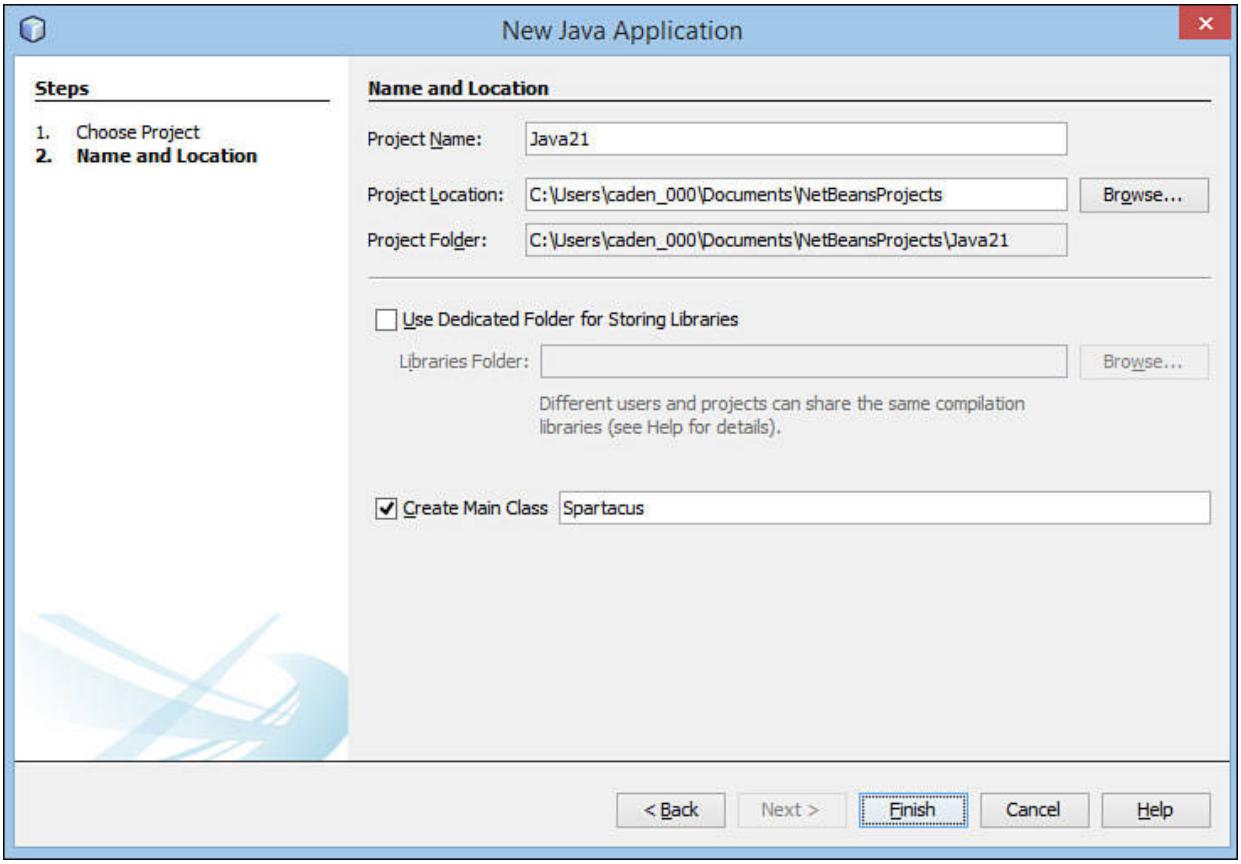


FIGURE A.2 The New Project Wizard.

NetBeans can create several types of Java projects, but during this book you can focus on just one: Java Application.

For your first project (and most of the projects in this book), choose the Java category and the project type Java Application; then click Next. The wizard asks you to choose a name and location for the project.

The Project Location text field identifies the root folder of the programming projects you create with NetBeans. In Windows, this is a subfolder of My Documents called NetBeansProjects. All projects you create are stored inside this folder, each in its own subfolder.

In the Project Name text field, enter `Java21`. The Create Main Class text box changes in response to the input, recommending `java21.Java21` as the name of the main Java class in the project. Change this to `Spartacus` and click Finish, accepting all other defaults. NetBeans creates the project and its first class.

Creating a New Java Class

When NetBeans creates a new project, it sets up all the necessary files and folders and creates starting code for the main class. Figure A.3 shows the first class in your project, `Spartacus.java`, open in the source editor.

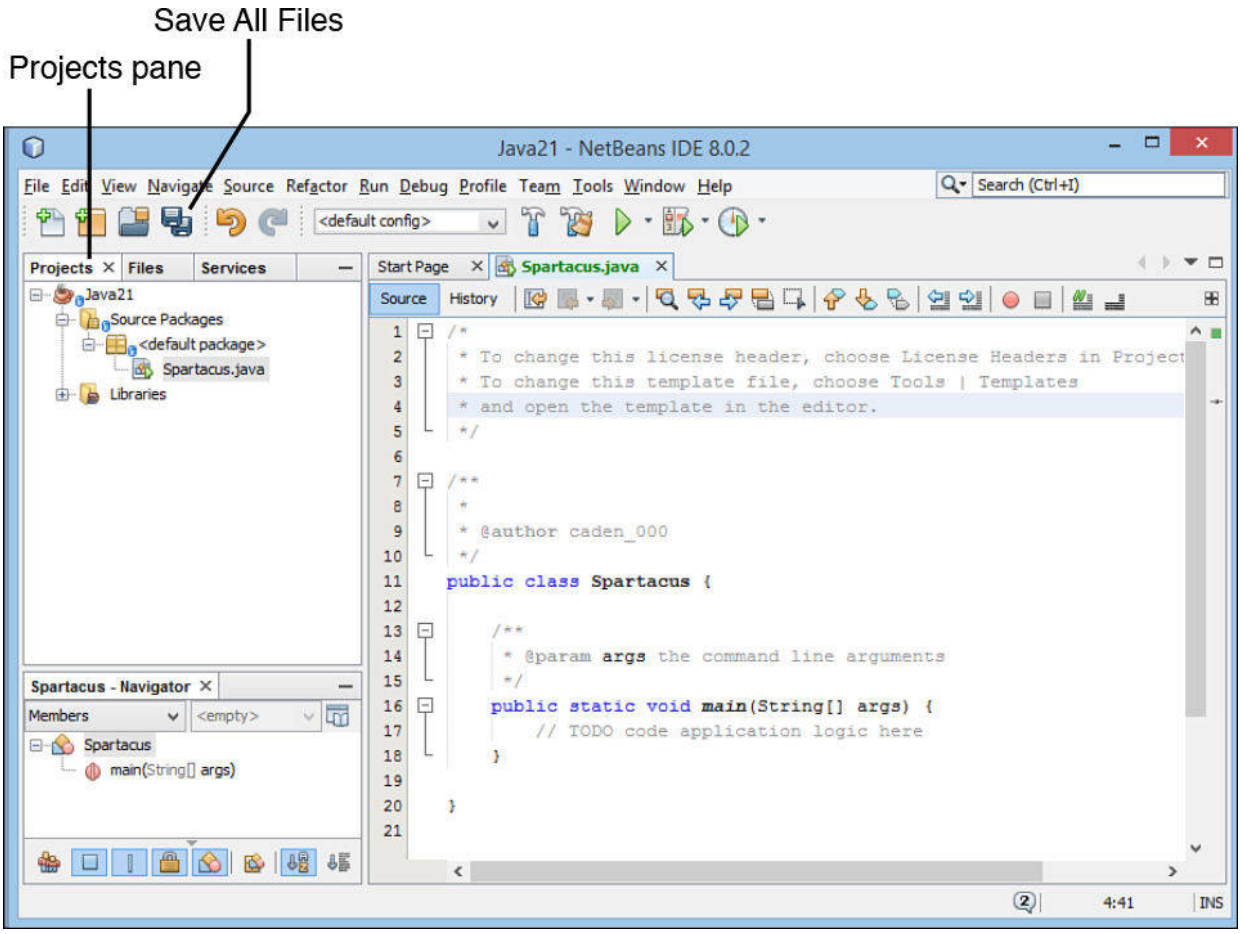


FIGURE A.3 The NetBeans source editor.

`Spartacus.java` is a bare-bones Java class that consists of only a `main()` method. All the light gray lines of code in the class are comments that exist to explain the class's purpose and function. Comments are ignored when the class is run.

To make the new class do something, add the following line of code on a new line right below the comment `// TODO code application logic here`:

Click here to view code image

```
System.out.println("I am Spartacus!");
```

The method `System.out.println()` displays a string of text—in this case, the sentence "I am Spartacus!"

Be sure to enter this code exactly as it is shown here. As you type, the source editor figures out what you're doing and displays helpful information related to the `System` class, the `out` instance variable, and the `println()` method. You'll love this stuff later, but for now, try your best to ignore it.

After you ensure that you typed the line correctly and ended it with a semicolon, click the Save All Files button on the toolbar to save the class.

Java classes must be compiled into executable bytecode before you can run them. This bytecode will be run by an interpreter called the Java Virtual Machine (JVM). NetBeans tries to compile classes automatically. You also can manually compile this class in two ways:

- Select Run, Compile File.
- Right-click `Spartacus.java` in the Projects pane to open a pop-up menu, and choose Compile File.

If NetBeans doesn't allow you to choose either of these options, NetBeans already has compiled the class.

If the class does not compile successfully, a white exclamation point in a red circle appears next to the filename `Spartacus.java` in the Projects pane. To fix the error, compare what you've typed in the text editor to the full source code of `Spartacus.java`, shown in Listing A.1, and resave the file.

LISTING A.1 The Full Text of `Spartacus.java`

Click here to view code image

```
 1: /*
 2:  * To change this template, choose Tools | Templates
 3:  * and open the template in the editor.
 4:  */
 5:
 6: /**
 7:  *
 8:  * @author User
 9:  */
10: public class Spartacus {
11:
12:     /**
13:      * @param args the command line arguments
14:      */
```

```
15:     public static void main(String[] args) {
16:         // TODO code application logic here
17:         System.out.println("I am Spartacus!");
18:
19:     }
20:
21: }
```

The class is defined in lines 10–21. Lines 1–9 are comments that NetBeans includes in every new class.

Running the Application

After you've created the Java application Spartacus and compiled it successfully, you can run it in the Java Virtual Machine within NetBeans in two ways:

- Choose Run, Run File.
- Right-click `Spartacus.java` in the Projects pane, and choose Run File.

When you run a Java class, the JVM calls its `main()` method. In the `Spartacus` class, the string "I am Spartacus!" appears in the Output pane, as shown in Figure A.4.

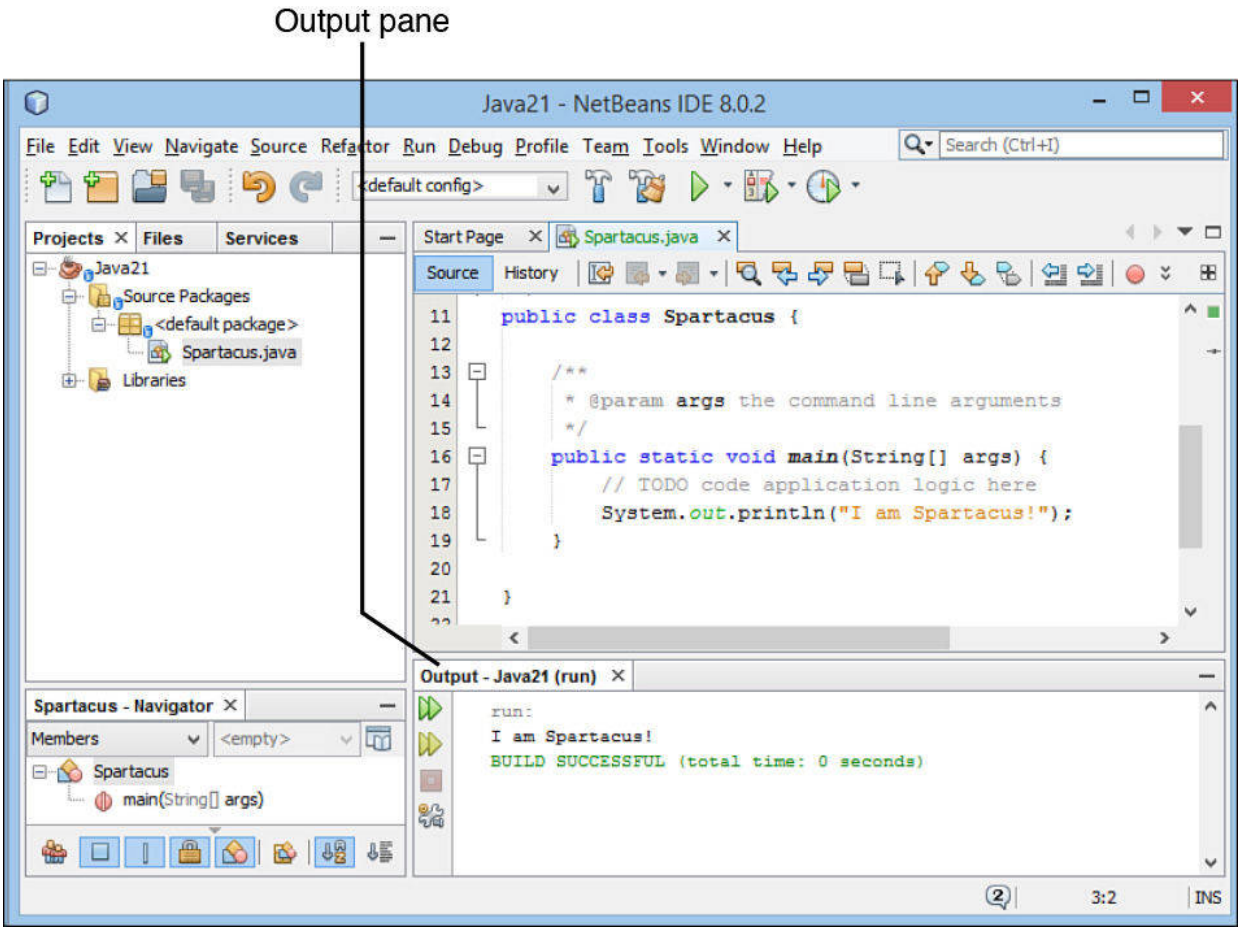


FIGURE A.4 Viewing program output in the NetBeans Output pane.

A Java class must have a `main()` method to be run. If you attempt to run a class that lacks a `main()` method, NetBeans responds with an error.

Fixing Errors

Now that the Spartacus application has been written, compiled, and run, it's time to break something to get some experience with how NetBeans responds when things go terribly wrong. Like any Java programmer, you'll soon get plenty of practice screwing up things on your own, but pay attention here anyway.

Return to `Spartacus.java` in the source editor, and remove the semicolon from the end of the line that calls `System.out.println()` (line 17 in Listing A.1). Even before you save the file, NetBeans spots the error and displays a red stop sign icon to the left of the line, as shown in Figure A.5.

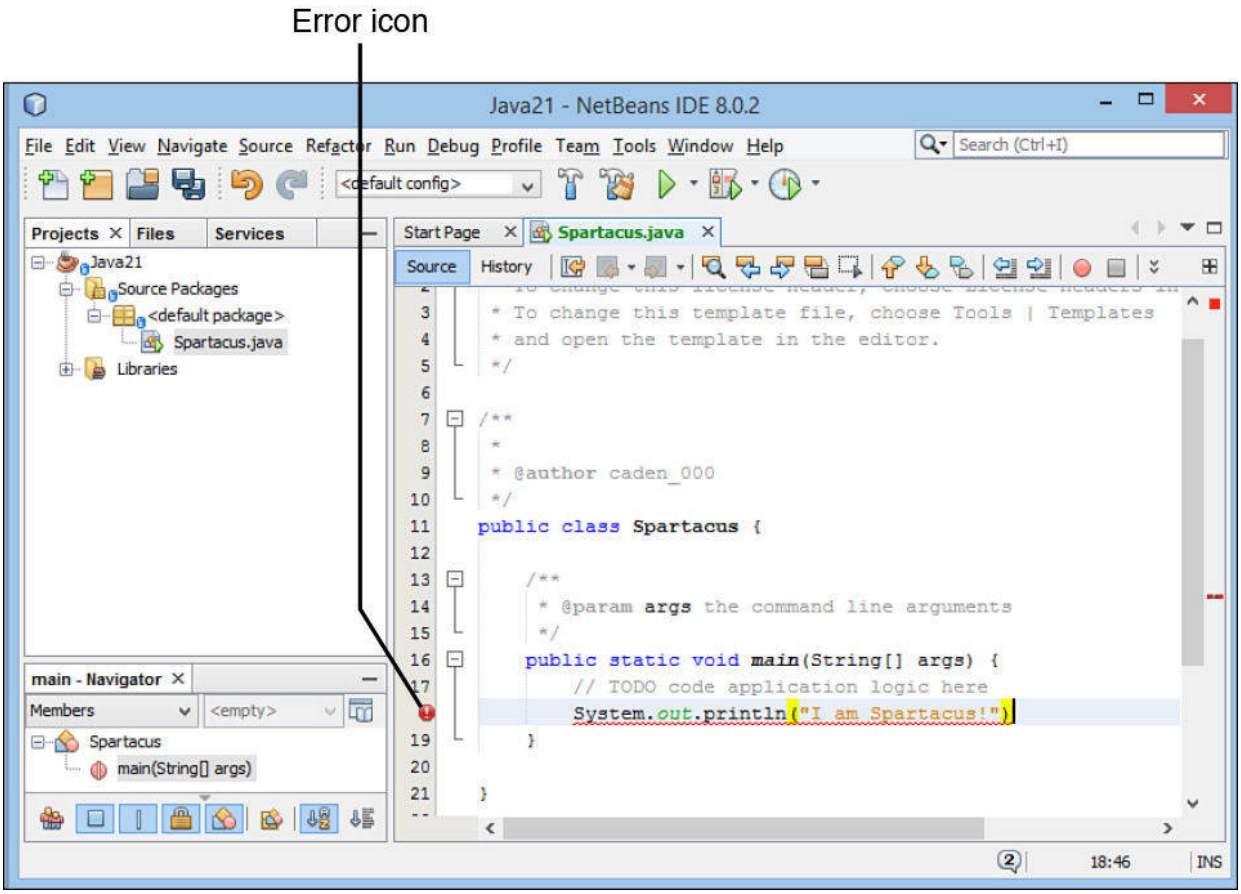


FIGURE A.5 Flagging errors in the source editor.

Hover the mouse cursor over the stop sign icon to see a dialog that describes the error NetBeans thinks it has spotted.

In this situation, the error message is simple: "';' expected."

The NetBeans source editor can identify many common programming errors and typos it encounters as you write a Java program. It stops the file from being compiled until the errors have been removed.

Put the semicolon back at the end of the line. The error icon disappears, and you can save and run the class again.

Expanding and Shrinking a Pane

As you use NetBeans, several panes usually will be open at the same time, including the source editor, Projects pane, and Output pane. They all compete for a limited amount of space on the program's user interface.

You can make one pane take up the entire NetBeans interface: Double-click the pane's tab.

To see this in action, double-click the tab `Spartacus.java`. The source editor expands, giving you more room to view the source code and make changes (Figure A.6).

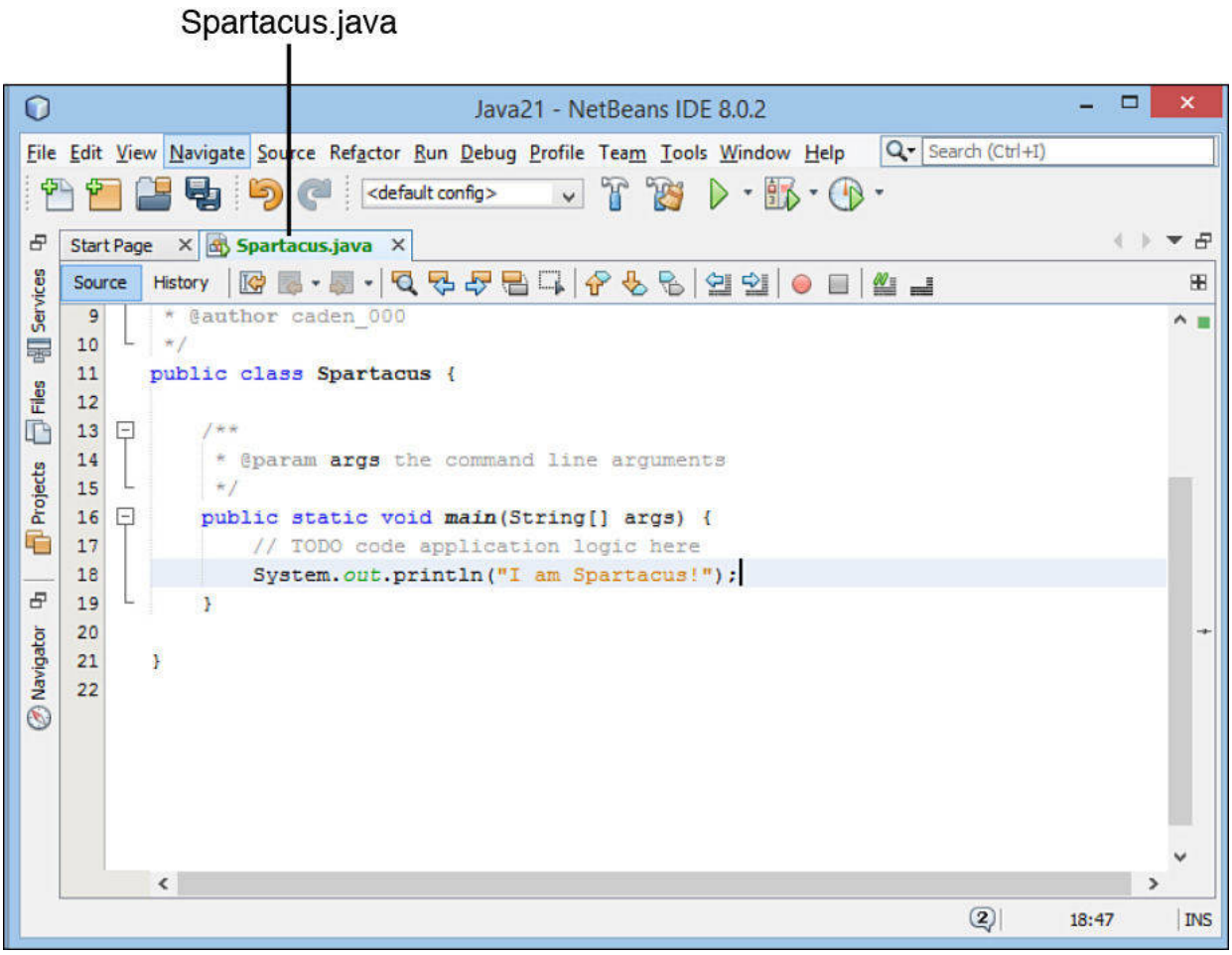


FIGURE A.6 Editing source code in a larger window.

The other panes close and are listed vertically along the left edge of the pane. Figure A.6 lists four panes: Navigator, Projects, Files, and Services.

To shrink the source editor and go back to the normal appearance of NetBeans, double-click the tab `Spartacus.java` again.

As you begin using NetBeans, it's common to accidentally expand a pane to fill the entire interface. You always can shrink it by double-clicking the pane's tab.

Exploring NetBeans

These basic features of NetBeans are all you need to create and compile the Java programs in this book.

NetBeans is capable of a lot more than the features described here, but you should focus on learning Java before diving too deeply into the IDE. Use NetBeans as if it were just a simple project manager and text editor. Write classes, flag errors, and make sure you can compile and run each project successfully.

When you're ready to learn more about NetBeans, Oracle offers training and documentation resources at www.netbeans.org/kb. You also will see links to the latest tutorials on the page that loads each time you start NetBeans.

Appendix B. This Book's Website

As much as I'd like to think otherwise, there are undoubtedly things you're unclear about after completing the 21 days of this book. Programming is a specialized, technical field that throws strange concepts and jargon at new learners, such as "instantiation," "ternary operators," and "big- and little-endian byte order."

If you have a question about any topic covered in the book, visit the book's website at www.java21days.com for assistance. Click the cover matching this edition of the book to visit its site.

The book's website offers the following:

- **Error corrections and clarifications**—When errors are brought to my attention, they are described on the site with the corrected text and any other material that will help.
- **Answers to reader questions**—If readers have questions that aren't covered in this book's Q&A sections, they may be presented on the site.
- **Sample files**—The source code and class files for all the programs you create during the book are available.
- **Sample Java programs**—Working versions of the programs featured in this book are available.
- **End-of-chapter features**—Solutions, including source code, for activities suggested at the end of each day and the answers to each day's certification practice question are available.
- **Updated links to the sites mentioned in this book**—If sites mentioned in the book have moved to a new address, they are listed.

You can email me by visiting the book's website. Click the Feedback link to be taken to a page where you can send email directly from the website. I also have a Twitter account at @rcade where I can be contacted to talk about the book, Java programming, and a wide variety of other topics—including Minecraft, the Jacksonville Jaguars, Sheffield Wednesday, science fiction, and popes.

—Rogers Cadenhead

Appendix C. Fixing a Problem with the Android Studio Emulator

The free Android Studio integrated development environment (IDE) has become the official tool for creating Android apps since its release in 2014. You learned how to create mobile apps in Java with this IDE during Day 21, "Writing Android Apps with Java."

If you have read that chapter and have successfully run an app in an Android emulator, you don't need to read this appendix.

But if you couldn't make the emulator work at all, you're in the right place.

Problems Running an App

When you are working on an Android project in Android Studio and you want to run the app, you can choose the menu command Run, Run App.

This command opens a Choose Device dialog that asks for the device where the app should be executed. The device can be a real Android phone or tablet, if it's connected to your computer over a USB cord and configured to test apps. The device also can be an Android emulator.

The Android emulator can act like actual phones and tablets that run the mobile OS. A virtual device can be set up for multiple Android virtual devices. When you install Android Studio, there's just one choice, which currently is Nexus 5 API 23 x86. This emulates a Nexus 5 phone from LG running version 23 of the Android API on an x86 processor.

Some users experience problems running an Android app for the first time with an emulator in Android Studio. The emulator crashes with this ominous message:

Output ▶

Click here to view code image

```
ERROR: x86 emulation currently requires hardware acceleration!
Please ensure Intel HAXM is properly installed and usable. CPU
acceleration status: HAX kernel module is not installed!
```

This error occurs on Windows computers and indicates that they need a hardware acceleration program from Intel called the Hardware Accelerated Execution Manager (HAXM) before the emulator will work. This program can be downloaded in Android Studio, but you must install it outside of the IDE.

HAXM is a hardware virtualization engine for computers with Intel processors that speeds up Android development by making emulators run faster. One of the biggest bottlenecks in app programming for Android is how slowly emulators load.

Before you set up HAXM, you must add it to the Android SDK in Android Studio.

Caution

HAXM only should be installed on computers with an Intel processor. This appendix resolves a problem where Android Studio indicates that it needs HAXM to run the Android emulator. If the emulator is failing with an error message that does not mention HAXM, don't use this appendix to fix it.

Install HAXM in Android Studio

HAXM can be downloaded and added to the Android SDK as you're running Android Studio. Click the Android SDK Manager button in the Android Studio toolbar, which is identified in Figure C.1.

FIGURE C.1 Running the Android SDK Manager.

The Android SDK Manager is used to enhance the SDK with additional versions of Android and useful SDK tools. Click the SDK Tools tab to bring it to the front.

The tools available for the SDK will be listed along with a check mark next to the ones you already have installed. Look for the item Intel x86 Emulator Accelerator (HAXM Installer).

If there's no check mark next to this item, it hasn't been added to the Android SDK in your copy of Android Studio. (If there is a check mark, it already has

been installed, so you should proceed to the next section, "Install HAXM on Your Computer.")

The Android SDK Manager is shown in Figure C.2.

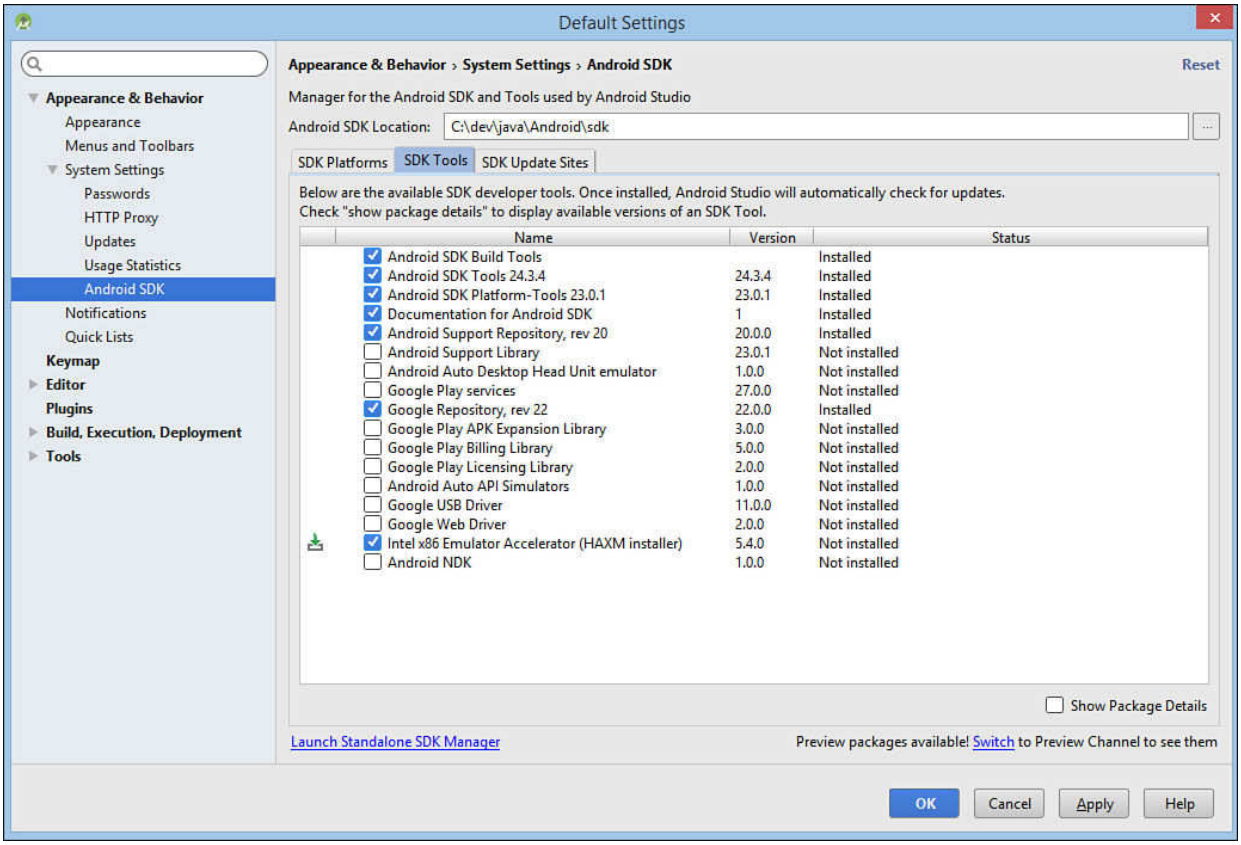


FIGURE C.2 Running the Android SDK Manager.

Select Intel x86 Emulator Accelerator (HAXM Installer) and click OK. You will be asked to confirm this change. Click OK.

Android Studio will download HAXM and report its progress. If it installs correctly, you're ready to proceed to the next step and install it on your computer.

Install HAXM on Your Computer

To begin setting up HAXM on your computer, first close Android Studio.

In your file system, find the folder where you told the Android Studio installation wizard to store the Android SDK.

If you don't remember where you put it, the default on Windows is to put the SDK in your personal user folder in a subfolder called `AppData\Local\Android\sdk`. So if your Windows username is

Catniss, the suggested SDK location is `\Users\Catniss\AppData\Local\Android\sdk`.

If you find the SDK's folder on your computer, open that folder, and then open the subfolder `extra\intel\Hardware_Accelerated_Execution_Manager`.

The folder contains a program called `intelhaxm-android`. This is the HAXM installation program.

HAXM requires a computer with an Intel processor, 1GB of disk space, and a Windows 7 or later, Windows Vista, or the 64-bit Mac OS X versions 10.8 through 10.10. A text file called Release Notes in this folder contains detailed information on the software requirements.

After you've reviewed the Release Notes file, if you are ready to install HAXM, run the program `intelhaxm-android`. The installer checks whether your computer can run HAXM and exits if it can't.

During installation you'll be asked how much memory to allow HAXM to use. The default value of 2GB should be sufficient. Complete the installation.

Tip

If you decide later that you've allocated too much or too little memory for HAXM, you can change this setting by running the installation program again.

After HAXM has been installed, you should reboot your computer.

When that's complete, load Android Studio and try to run your Android app again by choosing Run, Run App.

The app should run in an emulator. The Android emulator looks like a phone, displays an "Android" boot screen while it's loading, and then runs the app. Figure C.3 shows what the Palindrome app looks like when it has been run successfully.

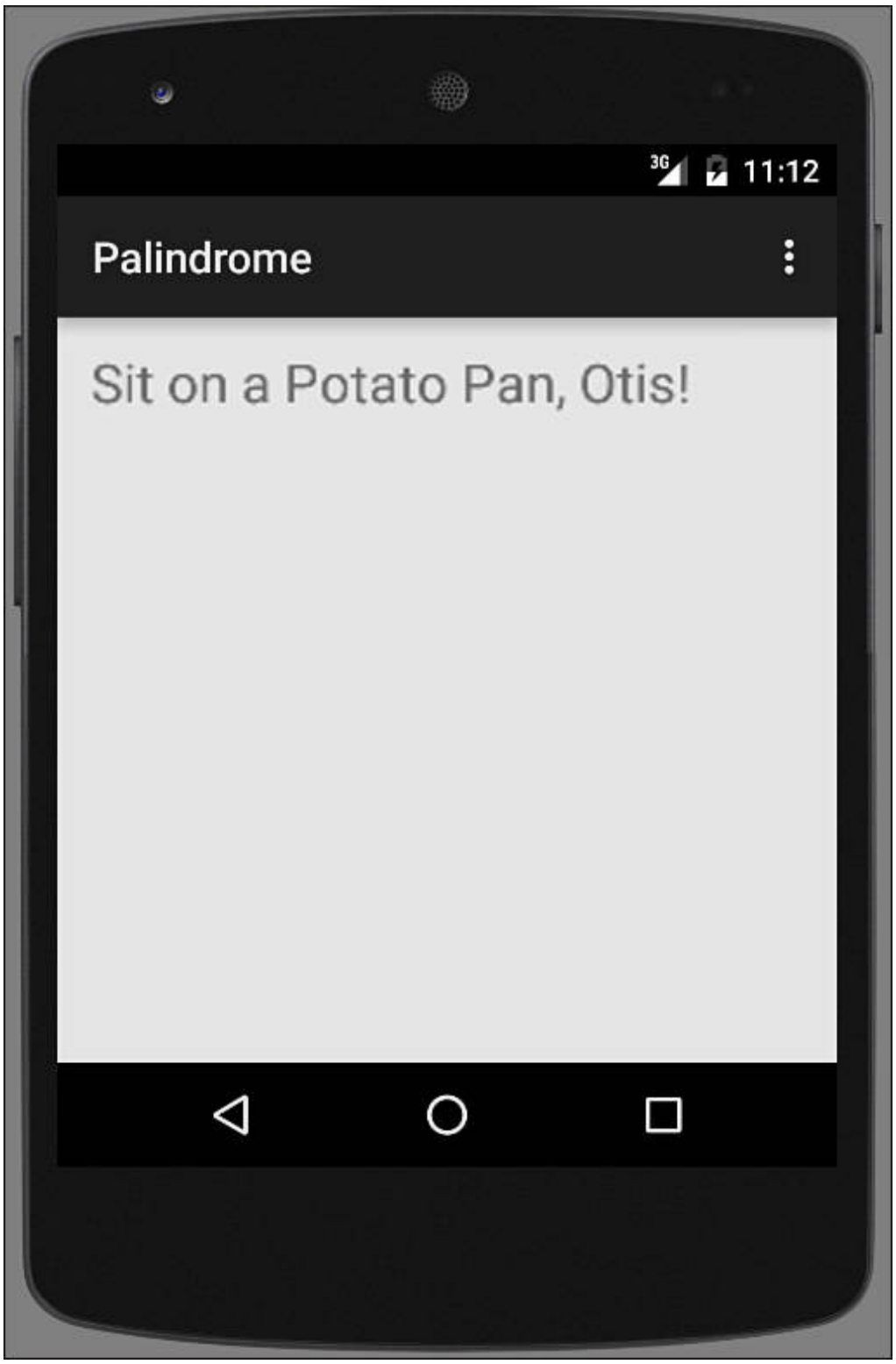


FIGURE C.3 Success! The emulator loads and runs an app.

If it worked, you're ready to go back to Day 21.

If it failed with the same error message asking you to "ensure Intel HAXM is properly installed and usable," there's one more thing you can check. However, it requires checking your computer's BIOS settings and making changes to them.

Checking BIOS Settings

For HAXM to work, your computer's BIOS must have Intel Virtualization Technology enabled in its settings. If you are an experienced computer user who is comfortable making changes to BIOS, this is a straightforward thing to check and change.

Because changes to BIOS can affect how your computer boots—or even stop it from booting Windows at all—you should poke around in BIOS only if you have made BIOS changes to a computer before. Otherwise, you should recruit the help of someone else who is an expert and can guide you through the process.

BIOS is the software that controls a Windows computer when you turn it on, taking care of booting the computer and other necessary hardware functions.

While your computer is booting, you briefly see a message about hitting a function key to check your BIOS settings.

If you don't hit this key, BIOS completes its work and Windows loads.

If you do hit the key, you see a screen like the one in Figure C.4.

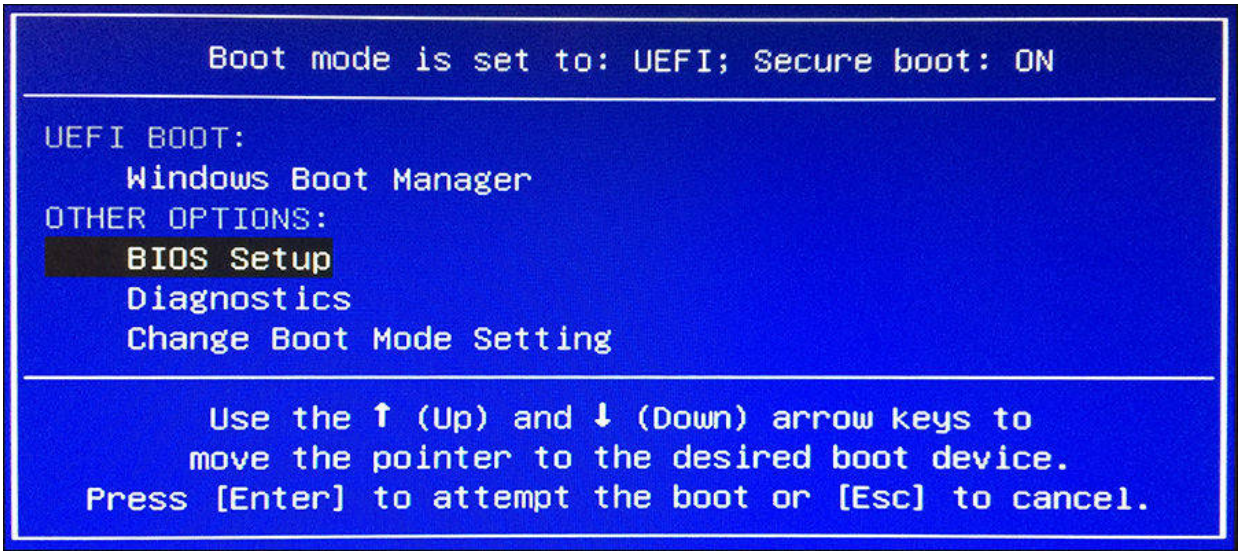


FIGURE C.4 Looking at a computer's BIOS settings.

Your computer's main menu for BIOS might look different from the one in Figure C.4. It varies depending on your computer manufacturer and the version of BIOS it uses.

On my Dell PC, I found out whether Intel Virtualization Technology was enabled by choosing BIOS Setup, Advanced, Processor Configuration. A list of processor settings was presented with [Enabled] or [Disabled] next to each one. These could be toggled from one setting to the other.

If you enable Intel Virtualization Technology in BIOS and save the change, your computer should be able to run HAXM, and the emulator problem should be resolved.

Appendix D. Using the Java Development Kit

In addition to the integrated development environment NetBeans, Oracle offers the Java Development Kit (JDK), a free set of command-line programs that are used to create, compile, and run Java programs. Every new release of Java is accompanied by a new release of the development kit. The current version is JDK version 8.

Although NetBeans and other programs such as IntelliJ IDEA and Eclipse are more sophisticated, some programmers continue to use the Java Development Kit. This appendix covers how to download and install the Java Development Kit, set it up on your computer, and use it to create, compile, and run a simple Java program.

It also describes how to fix a common configuration problem faced by JDK users.

Choosing a Java Development Tool

If you're using a Microsoft Windows or Apple Mac OS system, you probably have a Java Virtual Machine (JVM) installed that can run Java programs.

To develop Java programs, you need more than a JVM. You also need a compiler and other tools that are used to create, run, and test programs.

The Java Development Kit includes a compiler, JVM, debugger, file archiving program, and several other programs.

The kit is simpler than other development tools. It does not offer a graphical user interface, text editor, or other features that many programmers rely on.

To use the kit, you type commands at a text prompt. MS-DOS, Linux, and UNIX users will be familiar with this prompt, which also is called a command line.

Here's an example of a command you might type while using the Java Development Kit:

```
javac RetrieveMail.java
```

This command tells the `javac` program—the Java compiler included with the kit—to read a source code file called `RetrieveMail.java` and

create one or more class files. These files contain compiled bytecode that a JVM can execute.

When `RetrieveMail.java` is compiled, one of the files will be named `RetrieveMail.class`. If the class file was set up to function as an application, a JVM can run it.

People who are comfortable with command-line environments will be at home using the Java Development Kit. Everyone else must become accustomed to the lack of a graphical point-and-click environment as they develop programs.

If you have NetBeans or another Java development tool compatible with Java 8, you don't need to use the Java Development Kit. Many different development tools can be used to create the tutorial programs in this book.

Installing the Java Development Kit

You can download the Java Development Kit from Oracle's Java website at www.oracle.com/technetwork/java.

The website's Downloads section offers links to several versions of the Java Development Kit. It also offers the NetBeans development environment and other products related to the language. The product you should download is in the Java Standard Edition (Java SE) and is called the Java Software Development Kit version 8.

The kit is available for Windows, Mac OS, Linux, and Solaris SPARC systems.

The kit requires a computer with a Pentium 2 processor that is 266MHz or faster, 128MB of memory, and 300MB of free disk space.

When you're looking for this product, you might find that the Java Development Kit's version number has a number after 8, such as "JDK 8.0." To fix bugs and address security problems, Oracle periodically issues new releases of the kit and numbers them with a period and digit after the main version number. Choose the most current version of JDK 8 that's offered, whether it's numbered 8.0, 8.1, 8.2, or higher.

Caution

Take care not to download two similarly named products from Oracle by mistake: the Java Runtime Environment (JRE) 8.0 or the Java Standard Edition 8.0 Source Release.

To set up the kit, you must download and run an installation program. On the Java website, after you choose the version of the kit that's designed for your operating system, you can download it as a single file.

After you have downloaded the file, you'll be ready to set up the kit.

Windows Installation

Before installing the kit, make sure that no other Java development tools are installed on your system (assuming, of course, that you don't need any other tool at the moment). Having more than one Java programming tool installed on your computer can often cause configuration problems with the kit.

To set up the program on a Windows system, double-click the installation file or choose Start from the Windows taskbar to find and run the file.

The installation wizard guides you through the process of installing the software. If you accept the terms and conditions for using the kit, you'll be asked where to install the program, as shown in Figure D.1.

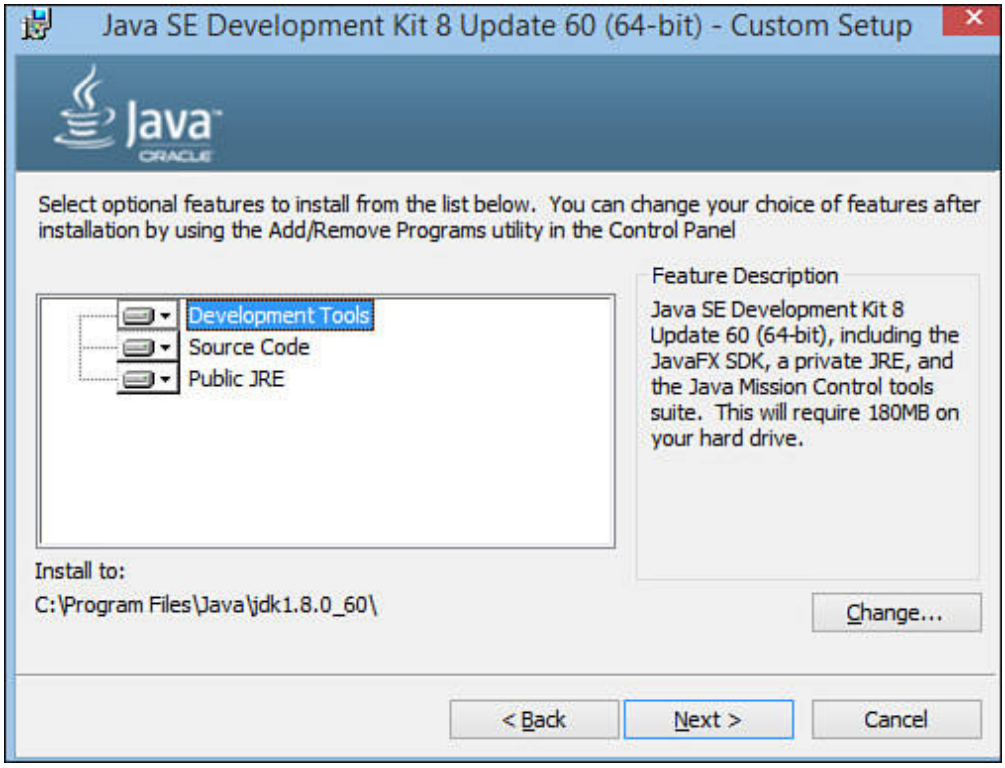


FIGURE D.1 Installing the JDK.

The wizard suggests a folder where the kit should be installed. In Figure D.1, the wizard suggests the folder `C:\Program Files\Java\jdk1.8.0_60`. When you install the kit, the suggested name might be different.

To choose a different folder, click the Change button. Either select or create a new folder, and click OK. The wizard returns to the Custom Setup options.

Tip

Before continuing, make note of the folder you have chosen. You'll need it later to configure the kit and fix any configuration problems that may occur.

You also are asked what parts of the kit to install. By default, the wizard installs all components of the JDK:

- **Development tools**—The executable programs needed to create Java software
- **Source code**—The source code for the thousands of classes that make up the Java Class Library

- **Public JRE**—A JVM you can distribute with the programs you create (also called a Java Runtime Environment)

If you accept the default installation, you need about 300MB of free hard disk space. You can save space by omitting everything but the program files. However, the source code and Java Runtime Environment can be useful, so it's a good idea to install them.

To prevent a component from being installed, click the hard drive icon next to its name, and then choose the This Feature Will Not Be Available option.

After you choose the components to install, click the Next button to continue. You may be asked whether to set up the Java Plug-in to work with the web browsers on your system.

The Java Plug-in is a JVM that runs Java programs incorporated into web pages. These programs, which are called applets, can work with different virtual machines, but most browsers do not include one that supports the current version of the Java language. Oracle offers the plug-in to provide full language support to Microsoft Internet Explorer, Microsoft Edge, Mozilla Firefox, Google Chrome, Safari, and other browsers.

After you complete the configuration, the wizard installs the kit on your system.

Configuring the Java Development Kit

After the wizard installs the kit, you must edit your computer's environment variables to include references to the kit.

Experienced MS-DOS users can finish setting up the kit by adjusting two variables:

- Edit the computer's `PATH` variable and add a reference to the kit's `bin` folder (which is `C:\Program Files\Java\jdk1.8.0_60\bin` if you installed the kit into the `C:\Program Files\Java\jdk1.8.0_60` folder).
- Edit or create a `CLASSPATH` variable so that it contains a reference to the current folder—a period and semicolon (`.;`)—followed by a reference to the `tools.jar` file in the kit's lib folder (which is `C:\Program Files\Java\jdk1.8.0_60\lib\tools.jar` if the kit was installed into `C:\Program Files\Java\jdk1.8.0_60`).

For Windows users unfamiliar with MS-DOS, later sections cover in detail how to set the `PATH` and `CLASSPATH` variables on a Windows system.

Users of other operating systems should follow the instructions provided by Oracle on its Java Development Kit download page.

Using a Command-Line Interface

The kit requires the use of a command line to compile Java programs, run them, and handle other tasks.

A command line is a way to operate a computer entirely by typing commands using the keyboard, rather than by using the mouse. Very few programs designed for Windows users require the command line today.

Note

To get to a command line in Windows, do the following:

- On Windows 8 and 10, choose Start, click the Search magnifying glass icon at upper right, enter `Command Prompt` in the search box, and click the Command Prompt icon.
- On Windows 7, Vista, XP, or Server 2003, choose Start, All Programs, Accessories, Command Prompt.
- On Windows 98 or Me, choose Start, Programs, MS-DOS Prompt.
- On Windows NT or 2000, choose Start, Programs, Accessories, Command Prompt.

When you open a command line in Windows, a new window opens in which you can type commands, as shown in Figure D.2.

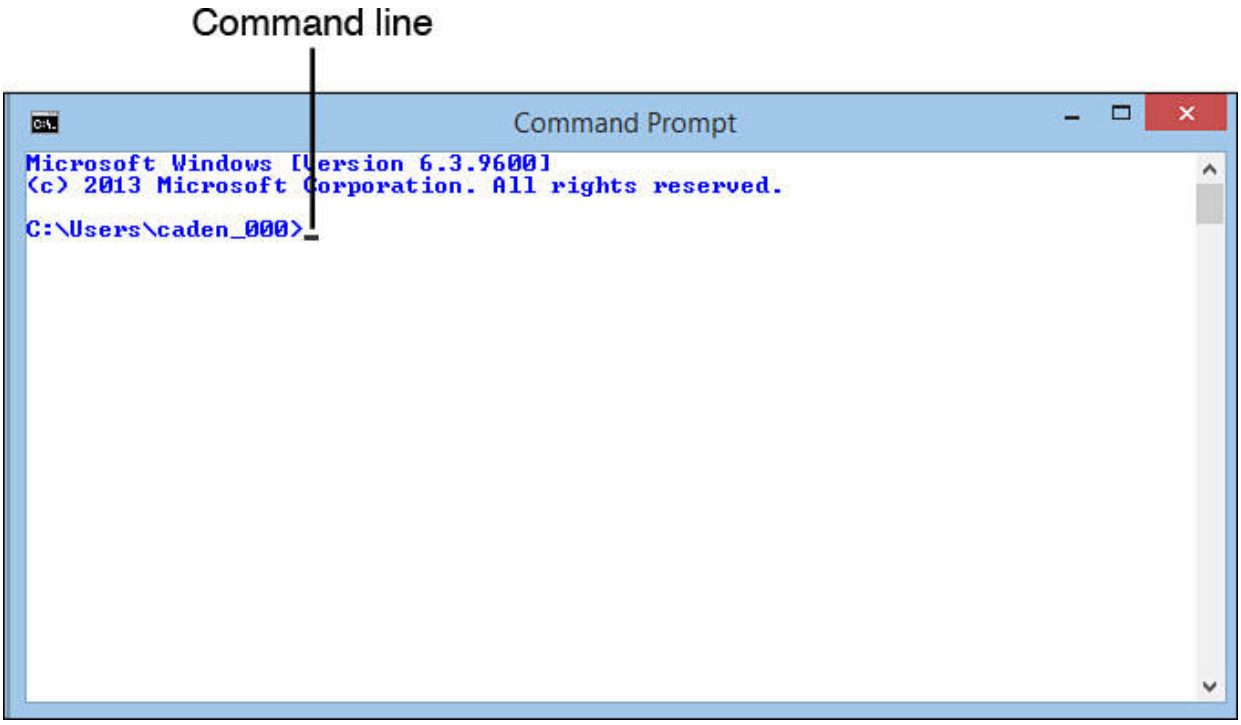


FIGURE D.2 Using a newly opened command-line window.

The command line in Windows uses commands adopted from MS-DOS, the Microsoft operating system that preceded Windows. MS-DOS supports the same functions as Windows—copying, moving, and deleting files and folders; running programs; scanning and repairing a hard drive; formatting a floppy disk; and so on.

In the window, a cursor blinks on the command line whenever you can type in a new command. In Figure D.2, `C:\Users\caden_000>` is the command line.

Because MS-DOS can be used to delete files and even format your hard drive, you should learn something about the operating system before experimenting with its commands.

Note

If you'd like to learn a lot about MS-DOS, a good book is *Special Edition Using MS-DOS 6.22*, 3rd Edition (ISBN 978-0-78972-573-8), published by Que. The emphasis is on the words "a lot," because this book is 1,056 pages long.

However, you need to know only a few things about MS-DOS to use the kit: how to create a folder, how to open a folder, and how to run a program.

Opening Folders in MS-DOS

When you are using MS-DOS on a Windows system, you have access to all the folders you normally use in Windows. For example, if you have a Windows folder on your C: hard drive, the same folder is accessible as `C:\Windows` from a command line.

To open a folder in MS-DOS, type the command `CD`, followed by the name of the folder, and press Enter. Here's an example:

```
CD C:\TEMP
```

When you enter this command, the TEMP folder on your system's C: drive is opened, if it exists. After you open a folder, the command line is updated with the name of that folder, as shown in Figure D.3.

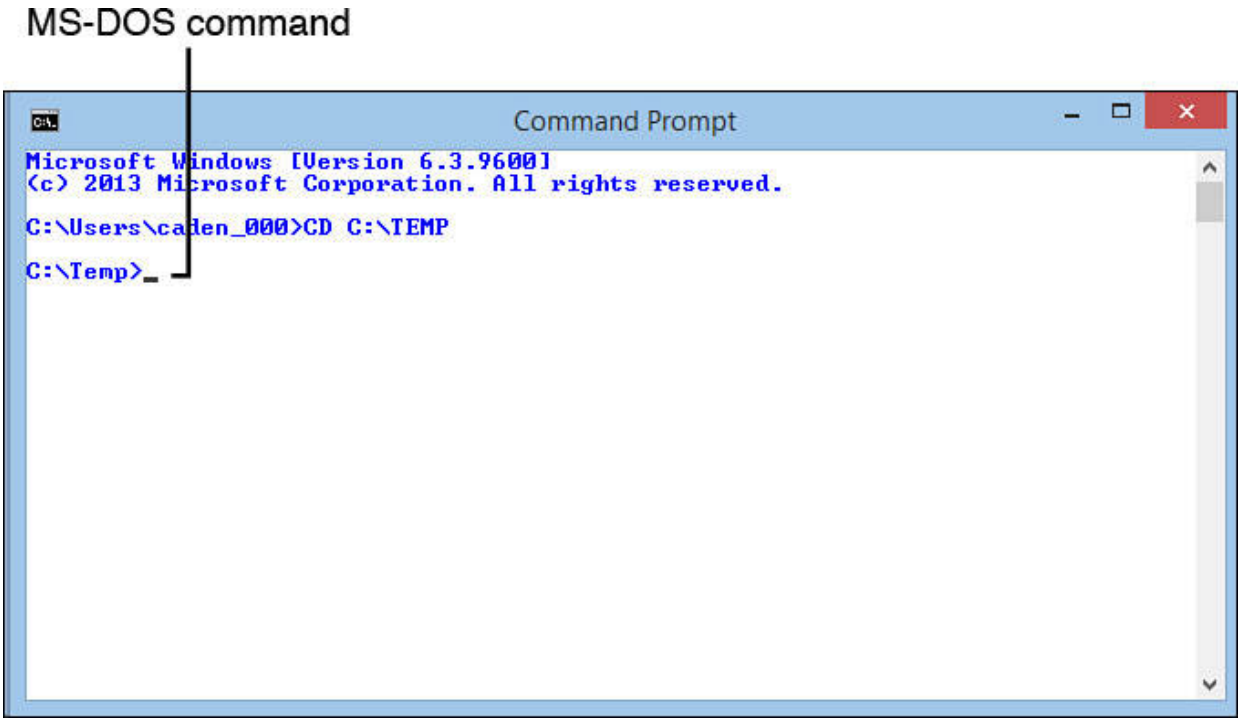


FIGURE D.3 Opening a folder in a command-line window.

You also can use the `CD` command in other ways:

- Type `CD \` to open the root folder on the current hard drive.
- Type `CD` *`foldername`* to open a subfolder matching the name you've used in place of *`foldername`*, if that subfolder exists.

- Type `CD ..` to open the folder that contains the current folder. For example, if you are in `C:\Windows\Fonts` and you use the `CD ..` command, `C:\Windows` is opened.

It's helpful to create a folder for the projects you create in this book, such as one named J21work. If you already have done this, you can switch to that folder by using the following commands:

```
CD \
CD J21work
```

If you haven't created that folder yet, you can do so using an MS-DOS command.

Creating Folders in MS-DOS

To create a folder from a command line, type the command `MD` followed by the folder's name, and press Enter, as in the following example:

```
MD C:\STUFF
```

The STUFF folder is created in the root folder of the system's C: drive. To open a newly created folder, use the `CD` command followed by that folder's name, as shown in Figure D.4.

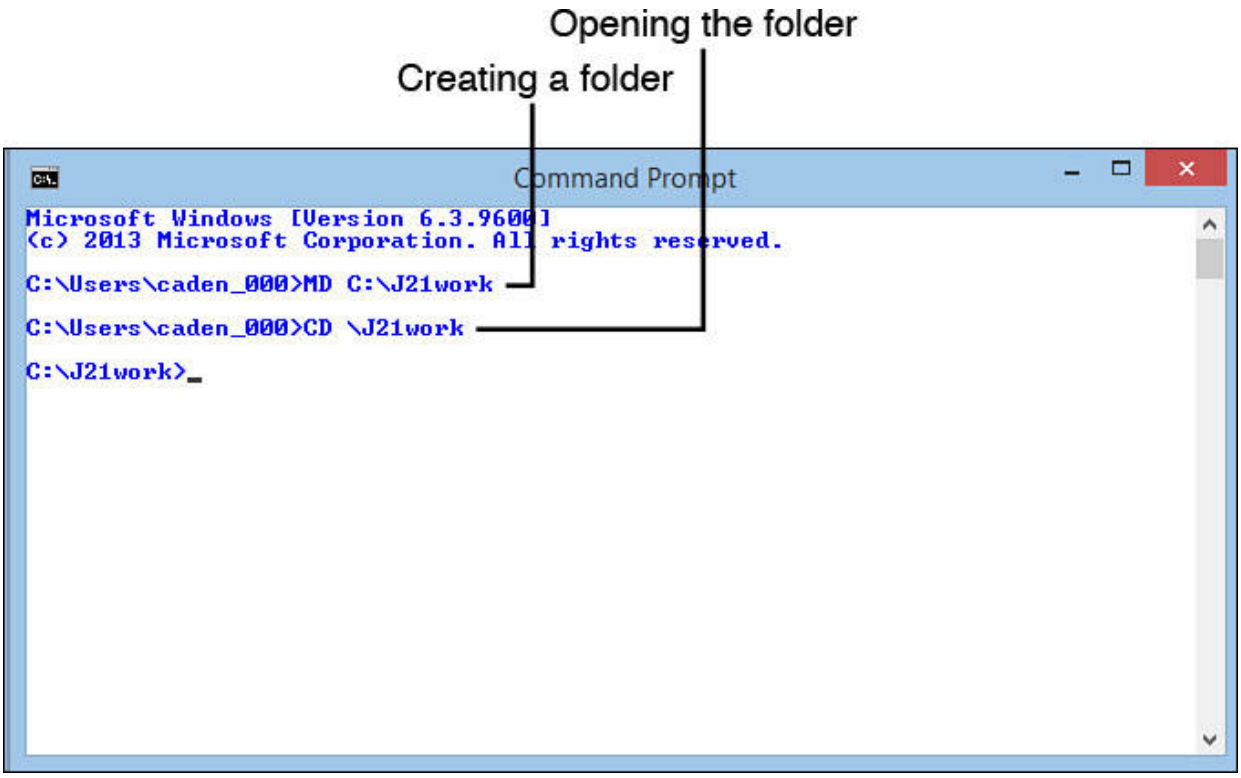


FIGURE D.4 Creating a new folder in a command-line window.

If you haven't already created a J21work folder, you can do so from a command line:

1. Change to the root folder (using the `CD \` command).
2. Type the command `MD J21work` and press Enter.

After J21work has been created, you can go to it at any time from a command line by using this command:

```
CD \J21work
```

The last thing you need to learn about MS-DOS to use the Java Development Kit is how to run programs.

Running Programs in MS-DOS

The simplest way to run a program at the command line is to type its name and press Enter. For example, type `DIR` and press Enter to see a list of files and subfolders in the current folder.

You also can run a program by typing its name followed by a space and some options that control how the program runs. These options are called *arguments*.

To see an example of this, change to the root folder (using `CD \`) and type `DIR J21work`. You'll see a list of files and subfolders contained in the J21work folder, if it contains any.

After you have installed the kit, run the JVM to see that it works. Type the following command at a command line:

```
java -version
```

`java` is the name of the JVM, and `-version` is an argument that tells it to display its version number.

You can see an example of this in Figure D.5, but your version number might be different, depending on what version of the kit you have installed.

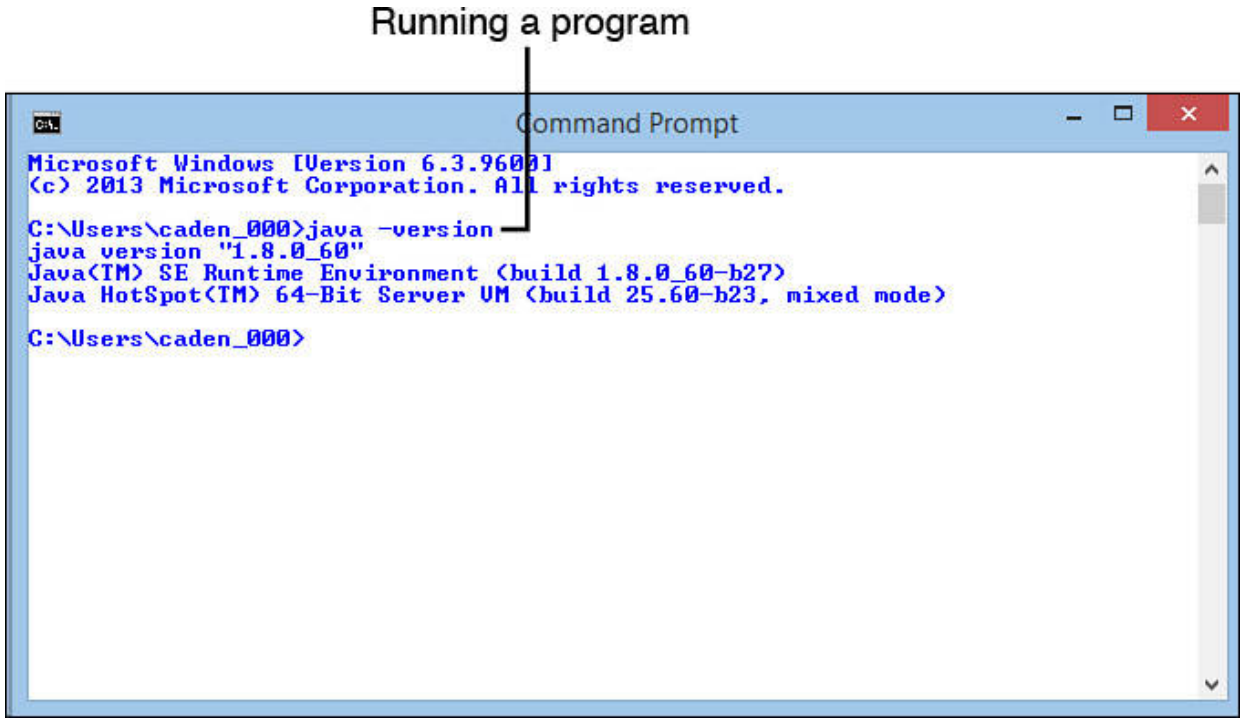


FIGURE D.5 Running the Java Virtual Machine in a command-line window.

If `java -version` works and you see a version number, it should begin with 1.8. Oracle tacks on a third number, but as long as the version number begins with 1.8, you are using the correct version of the Java Development Kit.

If you see an incorrect version number or a "Bad command or filename" error after running `java -version`, you need to make some changes to how the Java Development Kit is configured on your system.

Caution

Since Java 8 is the current version, you might be confused about the references to version 1.8.

Although the language is called Java 8 and the JDK is designated JDK 8.0, the kit's internal version number is 1.8.0. This internal number shows up in the `-version` command as well as in your choice of installation folder for the kit.

When all else fails, run the `java -version` command to make sure the right development tool has been installed on your system. If it begins with 1.8, you've got the right tool to develop programs for Java 8.

Correcting Configuration Errors

When you are writing Java programs for the first time, the most likely source of problems is not typos, syntax errors, or other programming mistakes. Most errors result from a misconfigured Java Development Kit.

If you type `java -version` at a command line and your system can't find the folder that contains `java.exe`, you see one of the following error messages or something similar (depending on your operating system):

- `Bad command or file name`
- `'java' is not recognized as an internal or external command, operable program, or batch file`

To correct this, you must configure your system's `PATH` variable.

Setting the `Path` on Most Windows Versions

On most versions of Windows, including Windows 7, 8, and 10, you configure the `Path` variable using the Environment Variables dialog, one of the features of the system's Control Panel.

To open this dialog on Windows 7, 8, or 10, follow these steps:

1. Click the Start button on the taskbar.
2. Click the Search icon (a magnifying glass) at the upper right.
3. Type `Environment Variables` in the search box.

4. Click the result Edit the System Environment Variables. The System Properties dialog opens with the Advanced tab in front.
5. Click the Environment Variables button. The Environment Variables dialog opens, as shown in Figure D.6.

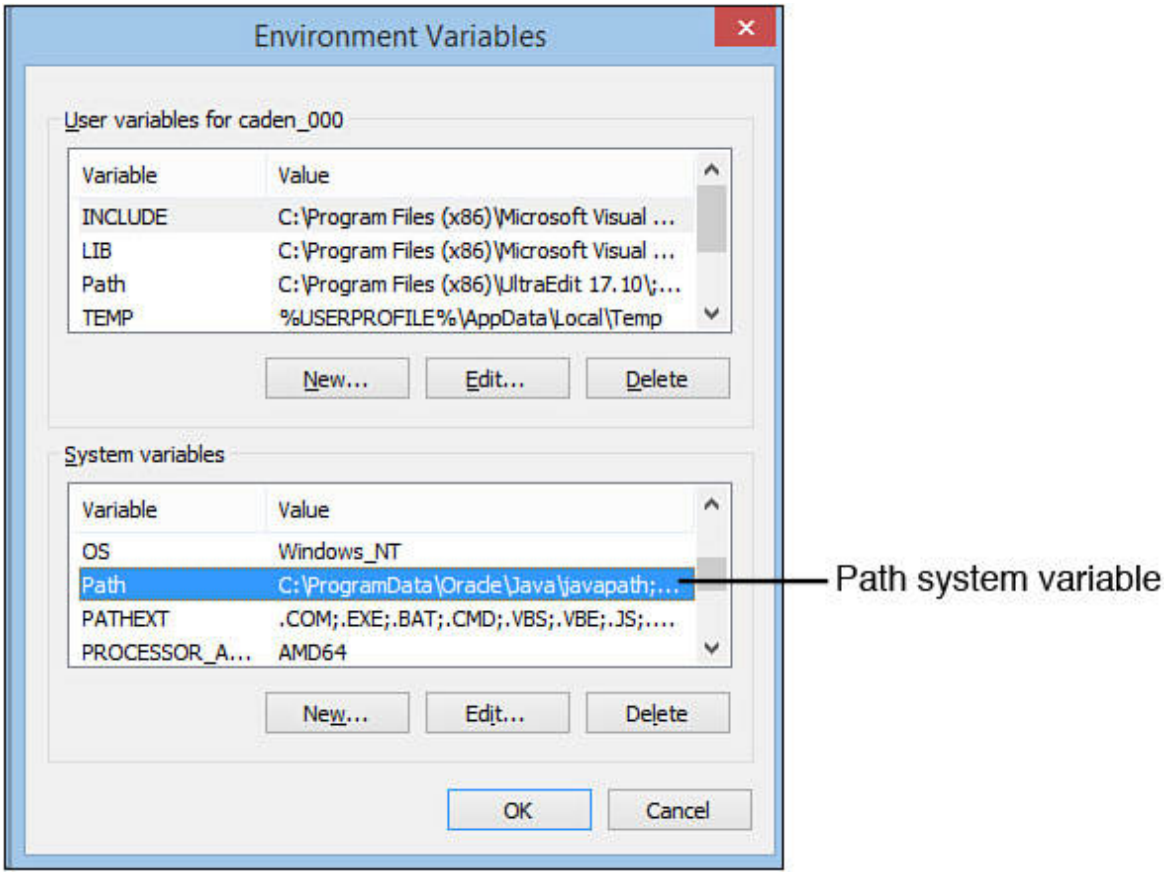


FIGURE D.6 Setting environment variables in Windows NT, XP, 2000, and 2003.

To open this dialog in other versions of Windows, follow these steps:

1. Right-click the Computer icon on your desktop or the Start menu, and choose Properties. The System Properties dialog opens.
2. Click the Advanced tab or the Advanced System Settings link.
3. Click the Environment Variables button. The Environment Variables dialog opens, as shown in Figure D.6.

You can edit two kinds of environment variables: system variables, which apply to all users on your computer, and user variables, which apply only to you.

`Path` is a system variable that helps MS-DOS find programs when you run them at a command line. It contains a list of folders separated by semicolons.

To set up the kit correctly, the folder that contains the Java Virtual Machine must be included in the `Path`. The virtual machine has the filename `java.exe`. If you installed the kit in the `C:\Program Files\Java\jdk1.8.0_60` folder on your system, `java.exe` is in `C:\Program Files\Java\jdk1.8.0_60\bin`.

If you can't remember where you installed the kit, you can look for `java.exe` by choosing Start, Search (or Start and the Search icon) and typing the filename in the search box. You might find several copies in different folders. To see which one is correct, open a command-line window and do the following for each copy you have found:

1. Use the `CD` command to open a folder that contains `java.exe`.
2. Run the command `java -version` in that folder.

When you know the correct folder, return to the Environment Variables dialog, select `Path` in the System Variables list, and click Edit. The Edit System Variable dialog opens with Path in the Variable Name field and a list of folders in the Variable Value field, as shown in Figure D.7.

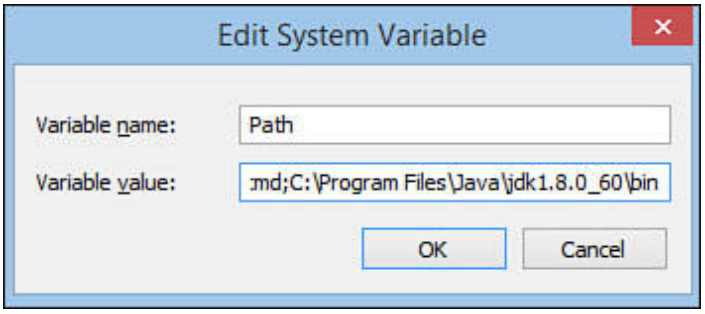


FIGURE D.7 Changing your system's `Path` variable.

To add a folder to the `Path`, click the Variable Value field and move the cursor to the end without changing anything. At the end, add a semicolon followed by the name of the folder that contains the Java Virtual Machine.

For example, if `C:\Program Files\Java\jdk1.8.0_60\bin` is the correct folder, add the following text to the end of the `Path` variable:

Click here to view code image

```
;c:\Program Files\Java\jdk1.8.0_60\bin
```

After making the change, click OK twice: once to close the Edit System Variable dialog, and another time to close the Environment Variables dialog.

Try it: Open a command-line window and type the command `java -version`.

If it displays the correct version of the Java Development Kit, your system is probably configured correctly, although you won't know for sure until you try to use the kit later in this appendix.

Setting the `PATH` on Windows 98 or Me

On a Windows 98 or Me system, you configure the `PATH` variable by editing the `AUTOEXEC.BAT` file in the root folder of your main hard drive. MS-DOS uses this file to set environment variables and configure how some command-line programs function.

`AUTOEXEC.BAT` is a text file you can edit with Windows Notepad. Start Notepad by choosing Start, Programs, Accessories, Notepad from the Windows taskbar.

The Notepad text editor opens. Choose File, Open from Notepad's menu bar, go to the root folder on your main hard drive, and then open the file `AUTOEXEC.BAT`.

When you open the file, you see a series of MS-DOS commands, each on its own line.

The only commands you need to look for are any that begin with `PATH`.

The `PATH` command is followed by a space and a series of folder names separated by semicolons. It sets up the `PATH` variable, a list of folders that contain command-line programs you use.

`PATH` helps MS-DOS find programs when you run them at a command line.

You can see what `PATH` has been set to by typing the following command at a command line:

```
PATH
```

To set up the kit correctly, the folder that contains the Java Virtual Machine must be included in the `PATH` command in `AUTOEXEC.BAT`.

The virtual machine has the filename `java.exe`. If you installed JDK 8 in the `C:\Program Files\Java\jdk1.8.0_60` folder on your system, `java.exe` is in `C:\Program Files\Java\jdk1.8.0_60\bin`.

If you can't remember where you installed the kit, you can look for `java.exe` by choosing Start, Find, Files or Folders. You might find several copies in different folders. To see which one is correct, open a command-line window and do the following for each copy you have found:

1. Use the `CD` command to open a folder that contains `java.exe`.
2. Run the command `java -version` in that folder.

When you know the correct folder, create a blank line at the bottom of the `AUTOEXEC.BAT` file and add the following:

```
PATH rightfoldername;%PATH%
```

For example, if `C:\Program Files\Java\jdk1.8.0_60\bin` is the correct folder, add the following line at the bottom of `AUTOEXEC.BAT`:

Click here to view code image

```
PATH c:\"Program Files"\Java\jdk1.8.0_60\bin;%PATH%
```

The addition of the text `%PATH%` keeps you from wiping out any other `PATH` commands in `AUTOEXEC.BAT`. Quotation marks appear around the folder name Program Files because some versions of Windows require this to handle folder names that contain spaces.

After making changes to `AUTOEXEC.BAT`, save the file and reboot your computer. When this is done, try the `java -version` command.

If it displays the correct version of the kit, your system is probably configured correctly. You'll find out for sure when you try to create a sample program later in this appendix.

Using a Text Editor

Unlike more sophisticated Java development tools such as NetBeans, the Java Development Kit does not include a text editor to use when you create source files.

For an editor or word processor to work with the kit, it must be able to save text files with no formatting.

This feature has different names in different editors. Look for a format option such as one of the following when you save a document or set the properties for a document:

- Plain text

- ASCII text
- DOS text
- Text-only

If you're using Windows, several editors are included with the operating system.

Windows Notepad is a no-frills text editor that works only with plain-text files. It can handle only one document at a time. On Windows 7, 8, or 10, choose Start, click the Search icon at the upper right, search for Notepad, and click the search result Notepad. On earlier versions of Windows, choose Start, Programs, Accessories, Notepad.

Windows WordPad is a step above Notepad. It can handle more than one document at a time and can handle both plain-text and Microsoft Word formats. It also remembers the last several documents it has worked on and makes them available from the File menu. It's on the Accessories menu along with Notepad.

Windows users also can use Microsoft Word, but you must save files as text rather than in Word's proprietary format. (UNIX and Linux users can author programs with emacs, pico, and vi; Macintosh users have SimpleText or any of the previously mentioned UNIX tools available for Java source file creation.)

One disadvantage of using simple text editors such as Notepad or WordPad is that they do not display line numbers as you edit.

Seeing the line number helps in Java programming because many compilers indicate the line number where an error occurred. Take a look at the following error generated by the JDK compiler:

Click here to view code image

```
Palindrome.java:2: Class Font not found in type declaration.
```

The number 2 after the name of the Java source file indicates the line that triggered the compiler error. With a text editor that supports numbering, you can go directly to that line and start looking for the error.

Usually there are better ways to debug a program with a Java integrated development environment. But kit users must search for compiler-generated errors using the line number indicated by the `javac` tool. This is one of the

best reasons to move on to an advanced Java development program after learning the language with the kit.

Tip

Another alternative is to use the kit with a programmer's text editor that offers line numbering and other features. One of the most popular for Java is jEdit, a free editor available for Windows, Linux, and other systems at www.jedit.org.

I use UltraEdit, an excellent programmer and web designer's editor that currently sells for $79.95. To find out more and download a trial version, visit www.ultraedit.com.

Creating a Sample Program

Now that you have installed and set up the Java Development Kit, you're ready to create a sample Java program to make sure it works.

Java programs begin as source code—a series of statements created using a text editor and saved as a text file. You can use any program you like to create these files, as long as it can save the file as plain, unformatted text.

The kit does not include a text editor, but most other Java development tools include a built-in editor for creating source code files.

Load your editor of choice and enter the Java program shown in Listing D.1. Be sure to correctly enter all the parentheses, braces, brackets, and quotation marks in the listing, and capitalize everything in the program exactly as shown. If your editor requires a filename before you start entering anything, call it `HelloUser.java`.

LISTING D.1 Source Code of `HelloUser.java`

Click here to view code image

```
1: public class HelloUser {
2:     public static void main(String[] arguments) {
3:         String username = System.getProperty("user.name");
4:         System.out.println("Hello " + username);
5:     }
6: }
```

The line numbers and colons at the beginning of each line are not part of the program. They're included so that I can refer to specific lines by number in each program as you read this book. If you're ever unsure about the source code of a program in this book, you can compare it to a copy on the book's official website at www.java21days.com.

After you finish typing in the program, save the file somewhere on your hard drive with the name `HelloUser.java`. Java source files must be saved with the extension `.java`.

Tip

If you have created a folder called J21work, you can save `HelloUser.java` in that folder. This makes it easier to find the file while using a command-line window.

If you're using Windows, a text editor such as Notepad might add an extra `.txt` file extension to the filename of any Java source files you save. For example, `HelloUser.java` is saved as `HelloUser.java.txt`. As a workaround to avoid this problem, place quotation marks around the filename when saving a source file.

Tip

A better solution is to permanently associate `.java` files with the text editor you'll be using. In Windows, open the folder that contains `HelloUser.java`, and double-click the file. If you have never opened a file with the `.java` extension, you're asked what program to use when opening files of this type. Choose your preferred editor, and select the option to make your choice permanent. From this point on, you can open a source file for editing by double-clicking the file.

The purpose of this project is to test the Java Development Kit. None of the Java programming concepts used in the six-line HelloUser program are described in this appendix.

You learn the basics of the language during the first several days of Week 1, "The Java Language." If you have figured out anything about Java simply by typing in Listing D.1, it's your own fault.

Compiling and Running the Program in Windows

Now you're ready to compile the source file with the kit's Java compiler, a program called `javac`. The compiler reads a `.java` source file and creates one or more `.class` files that can be run by a Java Virtual Machine.

Open a command-line window; then open the folder where you saved `HelloUser.java`.

If you saved the file in the `J21work` folder inside the root folder on your main hard drive, the following MS-DOS command opens the folder:

```
cd \J21work
```

When you are in the correct folder, you can compile `HelloUser.java` by entering the following at a command prompt:

```
javac HelloUser.java
```

Figure D.8 shows the MS-DOS commands used to switch to the \J21work folder and compile `HelloUser.java`.

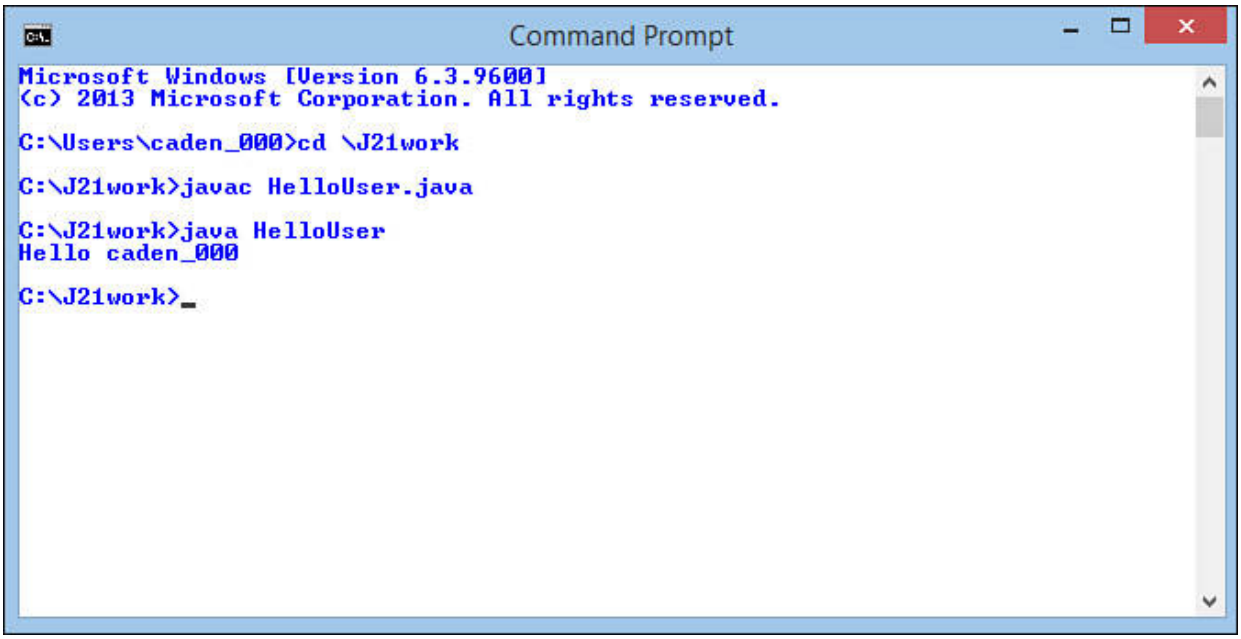


FIGURE D.8 Compiling a Java program in a command-line window.

The kit's compiler does not display a message if the program compiles successfully. If there are problems, the compiler tells you by displaying a message explaining each error along with the number of the line that triggered the error.

If the program compiles without any errors, a file called `HelloUser.class` is created in the same folder that contains `HelloUser.java`.

The class file contains the Java bytecode that a Java Virtual Machine will execute. If you get any errors, go back to your original source file and make sure that you typed it in exactly as it appears in Listing D.1.

After you have a class file, you can run that file using a JVM. The kit's JVM is called `java`, and it also is run from the command line.

Run the HelloUser program by switching to the folder containing `HelloUser.class` and entering the following:

```
java HelloUser
```

You see the text "Hello" followed by a space and your username.

Caution

When running a Java class with the kit's JVM, don't specify the `.class` file extension after the class's name. If you do, you'll see an error such as the following:

Click here to view code image

```
Exception in thread "main"
java.lang.NoClassDefFoundError: HelloUser/class
```

Figure D.8 shows the successful output of the HelloUser application along with the commands used to get to that point.

If you can compile the program and run it successfully, your kit is working, and you are ready to start Day 1, "Getting Started with Java."

If you cannot get the program to compile successfully even though you have typed it in exactly as it appears in the book, there may be one last problem with how the kit is configured on your system: The `CLASSPATH` environment variable might need to be configured.

Setting Up the `CLASSPATH` Variable

All the Java programs you write rely on two kinds of class files: the classes you create, and the Java Class Library, a set of hundreds of classes that represent the functionality of the Java language.

The kit needs to know where to find Java class files on your system. In many cases, the kit can figure this out on its own by looking in the folder where it was installed.

You also can set it up yourself by creating or modifying another environment variable: `CLASSPATH`.

Setting the `Classpath` on Most Windows Versions

If you have compiled and run the HelloUser program successfully, the kit has been configured successfully. You don't need to make any more changes to your system.

On the other hand, if you see a `Class not found` error or `NoClassDefFound` error whenever you try to run a program, you need to make sure your `Classpath` variable is set up correctly.

On most versions of Windows, including Windows 7 through 10, you configure the `Classpath` variable using the Environment Variables dialog. (Skip to the next section if you're using Windows 98 or Me.)

To open this dialog on Windows 7, 8, or 10, follow these steps:

1. Click the Start button on the taskbar.
2. Click the Search icon (a magnifying glass) at the upper right.
3. Type `Environment Variables` in the search box.
4. Click the result Edit the System Environment Variables. The System Properties dialog opens with the Advanced tab in front.
5. Click the Environment Variables button. The Environment Variables dialog opens, as shown in Figure D.9.

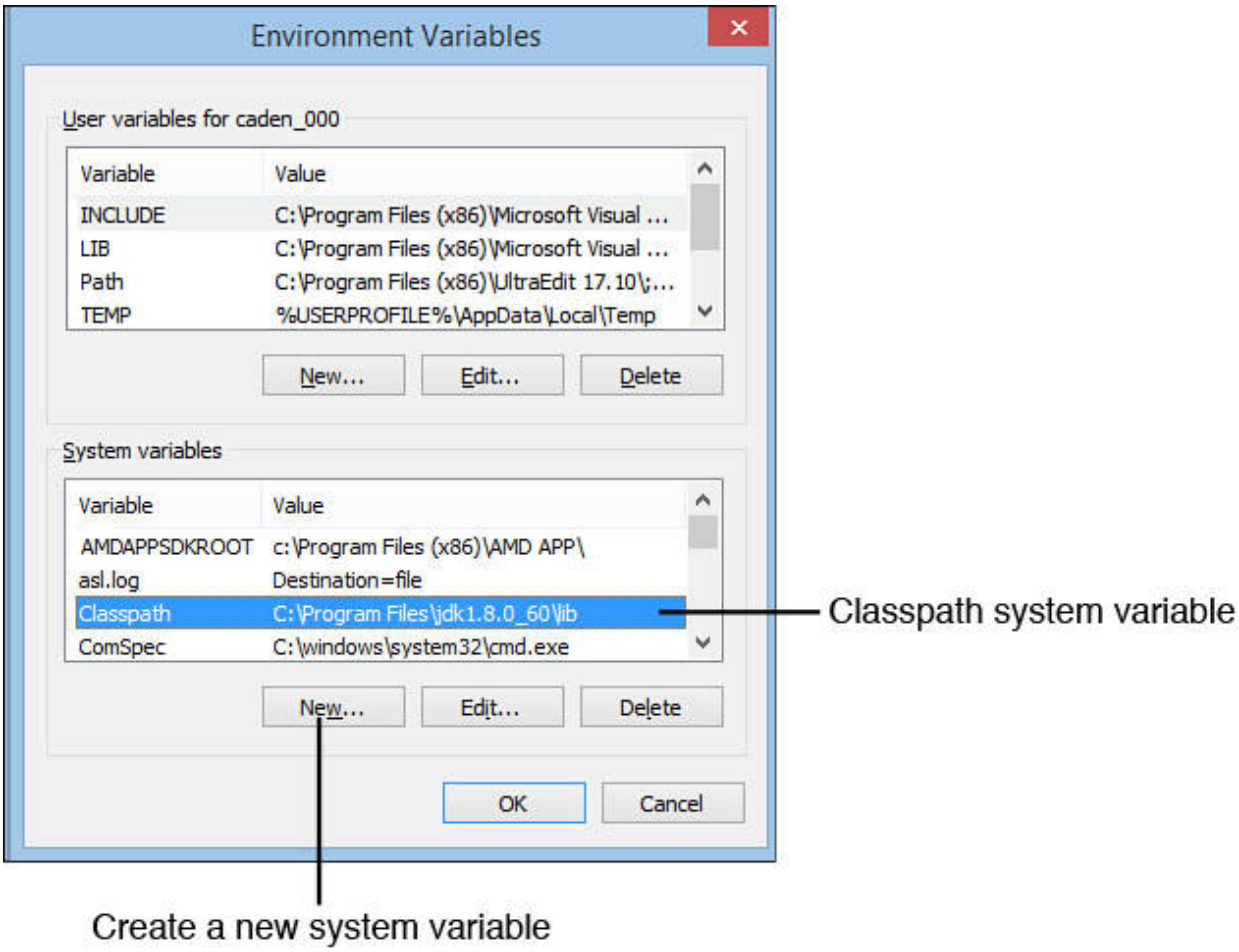


FIGURE D.9 Setting environment variables in most Windows versions.

To open it in other versions, do the following:

1. Right-click the My Computer icon on your desktop or Start menu, and choose Properties. The System Properties dialog opens.
2. Click the Advanced tab to bring it to the front.
3. Click the Environment Variables button. The Environment Variables dialog opens, seen in Figure D.9.

If your system has a `Classpath` variable, it probably is one of the system variables. Your system may not have a `Classpath` variable set. Normally the kit can find class files without the variable.

However, if your system has a `Classpath`, it must be set up with at least two things: a reference to the current folder (a period) and a reference to a file that contains the Java Class Library, `tools.jar`.

If you installed the kit in the `C:\Program Files\Java\jdk1.8.0_60` folder, `tools.jar` is in the folder

`C:\Program Files\Java\jdk1.8.0_60\lib`.

If you can't remember where you installed the kit, you can look for `tools.jar` by choosing Start, Search (or Start, Search icon) from the Windows taskbar. If you find several copies, you should be able to find the correct one using this method:

1. Use `CD` to open the folder that contains the JJVM (`java.exe`).
2. Enter the command `CD ..`.
3. Enter the command `CD lib`.

The lib folder normally contains the right copy of `tools.jar`.

When you know the correct folder, return to the Environment Variables dialog, shown in Figure D.9.

If your system does not have a `Classpath`, click the New button under the System Variables list. The New System Variable dialog opens.

If your system has a `Classpath`, choose it and click the Edit button. The Edit System Variable dialog opens.

Both dialogs contain the same thing: a Variable Name field and a Variable Value field.

Enter `Classpath` in the Variable Name field and the correct value for your `Classpath` in the Variable Value field.

For example, if you installed the kit in `C:\Program Files\Java\jdk1.8.0_60`, your `Classpath` should contain the following:

Click here to view code image

```
.;C:\"Program Files"\Java\jdk1.8.0_60\lib\tools.jar
```

After setting up your `Classpath`, click OK twice: once to close the Edit or New System Variable dialog, and again to close the Environment Variables dialog.

To see whether this change has fixed your problem, open a new command-line window and type the command `java -version`.

If it displays the correct version of the kit, your system should be configured correctly and require no more adjustments. Try creating the sample HelloUser program again. It should work after the `CLASSPATH` variable has been set up correctly.

Setting the CLASSPATH on Windows 98 or Me

If you are using Windows 98 or Me and the HelloUser program fails with a `Class not found` or `NoClassDefFound` error when you run it, you need to make sure your `CLASSPATH` variable is set up correctly.

To do this, run Windows Notepad and choose File, Open. Go to the root folder on your system, and then open the file `AUTOEXEC.BAT`. A file containing several MS-DOS commands is loaded in the editor.

Look for a line in the file that contains the text `SET CLASSPATH=` command followed by a series of folders and filenames separated by semicolons.

`CLASSPATH` is used to help the Java compiler find the class files it needs. A `CLASSPATH` can contain folders or files. It also can contain a period character (`.`), which is another way to refer to the current folder in MS-DOS.

You can see your system's `CLASSPATH` variable by typing the following command at a command line:

```
ECHO %CLASSPATH%
```

If your `CLASSPATH` includes folders or files that you know are no longer on your computer, you should remove the references to them on the `SET CLASSPATH=` line in `AUTOEXEC.BAT`. Be sure to remove any extra semicolons also.

To set up the kit correctly, you must include the file containing the Java Class Library in the `SET CLASSPATH=` command. This file has the filename `tools.jar`. If you installed the kit in the `C:\Program Files\Java\jdk1.8.0_60` folder on your system, `tools.jar` is probably in the folder `C:\Program Files\Java\jdk1.8.0_60\lib`.

If you can't remember where you installed the kit, you can look for `tools.jar` by choosing Start, Find, Files or choosing Folders from the Windows taskbar. If you find several copies, you should be able to find the correct one using this method:

1. Use `CD` to open the folder that contains the Java Virtual Machine (`java.exe`).
2. Enter the command `CD ..`.
3. Enter the command `CD lib`.

The lib folder normally contains the right copy of `tools.jar`.

When you know the correct location, create a blank line at the bottom of the AUTOEXEC.BAT file and add the following:

Click here to view code image

```
SET CLASSPATH=%CLASSPATH%;.;rightlocation
```

For example, if the tools.jar file is in the C:\Program Files\Java\jdk1.8.0_60\lib folder, add the following line at the bottom of AUTOEXEC.BAT:

Click here to view code image

```
SET CLASSPATH=%CLASSPATH%;.;c:\"Program
Files"\Java\jdk1.8.0_60\lib\tools.jar
```

After making changes to AUTOEXEC.BAT, save the file and reboot your computer. After this is done, try to compile and run the HelloUser sample program again. You should be able to accomplish this after the CLASSPATH variable has been set up correctly.

Appendix E. Programming with the Java Development Kit

The Java Development Kit (JDK) can be used throughout this book to create, compile, and run Java programs.

The tools that make up the kit contain numerous features that many programmers don't explore. Some of the tools themselves might be new to you.

This appendix covers features of the kit that you can use to create more reliable, better-tested, and faster-running Java programs.

Overview of the JDK

Although you can use numerous integrated development environments to create Java programs, the most widely used may still be the Java Development Kit (JDK) from Oracle, the set of command-line tools that are used to develop software with the Java language.

There are two main reasons for the kit's popularity:

- It's free. You can download a copy at no cost from the official Java website at www.oracle.com/technetwork/java.
- It's first. Whenever a new version of the language is released, the first tools that support the new version are in the kit.

The kit uses the command line. This is also called the MS-DOS prompt, command prompt, or console under Windows and the shell prompt under UNIX. You enter commands using the keyboard, as in this example:

```
javac VideoBook.java
```

This command compiles a Java program called `VideoBook.java` using the kit's compiler. The command has two elements: the name of the compiler, `javac`, and the name of the program to compile, `VideoBook.java`. A space character separates the two elements.

Each kit command follows the same format: the name of the tool to use, followed by one or more elements indicating what the tool should do. These elements are called *arguments*.

The following illustrates the use of command-line arguments:

Click here to view code image

```
java VideoBook add DVD "Broadcast News"
```

This command tells the Java Virtual Machine (JVM) to run a class file called `VideoBook` with three command-line arguments: the strings “add”, “DVD”, and “Broadcast News.”

Note

You might think there are more than three command-line arguments because of the spaces in the string “Broadcast News”. The quotation marks around that string cause it to be considered one command-line argument, which makes it possible to include spaces in an argument.

Some arguments used with the kit modify how a tool functions. These arguments are preceded by a hyphen character and are called *options*.

The following command shows the use of an option:

```
java -version
```

This command tells the JVM to display its version number rather than trying to run a class file. It’s a good way to find out whether the kit is correctly configured to run Java programs on your system. Here’s an example of the output run on a system equipped with Java 8:

Output ▶

Click here to view code image

```
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

The version reflects Oracle’s internal number for Java 8, which is 1.8.

In some instances, you can combine options with other arguments. For example, if you compile a Java class that uses deprecated methods, you can see more information on these methods by compiling the class with a `-deprecation` option, as in the following:

Click here to view code image

```
javac -deprecation OldVideoBook.java
```

The `java` Virtual Machine

`java`, the Java Virtual Machine, is used to run Java applications from the command line. It takes as an argument the name of a class file to run, as in the following example:

```
java BidMonitor
```

Although Java class files end with the `.class` extension, this extension is not specified when the JVM is used. The machine also is called the Java interpreter.

The class loaded by the JVM must contain a class method called `main()` that takes the following form:

Click here to view code image

```
public static void main(String[] arguments) {
    // method here
}
```

Some simple Java programs might consist of only one class—the one containing the `main()` method. In more complex programs that use other classes, the JVM automatically loads any other classes that are needed.

The JVM runs bytecode, compiled instructions that the machine executes. After a Java program is saved in bytecode as a `.class` file, it can be run by different JVMs without modification. If you have compiled a Java program, it will be compatible with any JVM that fully supports Java.

Note

Interestingly, Java is not the only language that you can use to create Java bytecode. NetRexx, JPython, JRuby, JudoScript, and several dozen other languages compile into `.class` files of executable bytecode through the use of compilers specific to those languages. Robert Tolksdorf maintains a comprehensive list of these languages at www.is-research.de/info/vmlanguages.

You can specify the class file that the JVM will run in two different ways. If the class is not part of any package, you can run it by specifying the class's name, as in the preceding `java BidMonitor` example. If the class is part

of a package, you must specify the class by using its full package and class name.

For example, consider a `SellItem` class that is part of the `org.cadenhead.auction` package. To run this application, you would use the following command:

Click here to view code image

```
java org.cadenhead.auction.SellItem
```

Each element of the package name corresponds to its own subfolder. The JVM looks for the `SellItem.class` file in several different places:

- The `org\cadenhead\auction` subfolder of the folder where the `java` command was entered (If the command was entered from the `C:\J21work` folder, for example, the `SellItem.class` file can be run successfully if it is in the `C:\J21work\org\cadenhead\auction` folder.)
- The `org\cadenhead\auction` subfolder of any folder in your `Classpath` setting

If you're creating your own packages, an easy way to manage them is to add a folder to your `Classpath` that's the root folder for any packages you create, such as `C:\javapackages` or something similar. After creating subfolders that correspond to the name of a package, place the package's class files in the correct subfolder.

You can specify a `Classpath` when running a Java application with the command-line option `-cp`. Here's an example:

Click here to view code image

```
java -cp . org.cadenhead.auction.SellItem
```

This command sets the `Classpath` to ".", which represents the current folder.

Java supports assertions, a debugging feature that works only when requested as a command-line option. To run a program using the JVM and make use of any assertions it contains, use the command-line option `-ea`, as in the following example:

```
java -ea Outline
```

The JVM executes all `assert` statements in the application's class and all other class files it uses, with the exception of classes from the Java Class Library.

To remove that exception and make use of all assertions, run a class with the `-esa` option.

If you don't specify one of the options that turns on the assertions feature, the JVM ignores all `assert` statements.

The `javac` Compiler

The Java compiler, `javac`, converts Java source code into one or more class files of bytecode that a JVM can run.

Java source code is stored in a file with the `.java` file extension. This file can be created with any text editor that can save a document without any special formatting codes. The terminology varies depending on the text-editing software being used, but these files are often called plain text, ASCII text, DOS text, or something similar.

A Java source code file can contain more than one class, but only one of the classes can be declared to be public. A class can contain no public classes at all if desired, although this isn't possible with applets because of the rules of inheritance.

If a source code file contains a class that has been declared to be public, the filename must match the name of that class. For example, the source code for a public class called `BuyItem` must be stored in a file called `BuyItem.java`.

To compile a file, you run the `javac` tool with the name of the source code file as an argument, as in the following:

```
javac BuyItem.java
```

You can compile more than one source file by including each separate filename as a command-line argument, such as this command:

Click here to view code image

```
javac BuyItem.java SellItem.java
```

You also can use wildcard characters such as `*` and `?`. Use the following command to compile all `.java` files in a folder:

```
javac *.java
```

When you compile one or more Java source code files, a separate `.class` file is created for each Java class that compiles successfully.

Another useful option when running the compiler is `-deprecation`, which causes the compiler to describe any deprecated methods that are being employed in a Java program.

A deprecated method is one that Oracle has replaced with a better alternative, either in the same class or in a different class. Although the deprecated method works, at some point Oracle may decide to remove it from the class. The deprecation warning is a strong suggestion to stop using that method as soon as you can.

Normally, the compiler issues a single warning if it finds any deprecated methods in a program. The `-deprecation` option causes the compiler to list each method that has been deprecated, as in the following command:

Click here to view code image

```
javac -deprecation SellItem.java
```

If you're more concerned with the speed of a Java program than the size of its class files, you can compile its source code with the `-O` option. This creates class files that have been optimized for faster performance. Methods that are static, final, or private might be compiled inline, a technique that makes the class file larger but causes the methods to be executed more quickly.

If you plan to use a debugger to look for bugs in a Java class, compile the source with the `-g` option to put all debugging information in the class file, including references to line numbers, local variables, and source code. (To keep all this out of a class, compile with the `-g:none` option.)

Normally, the Java compiler doesn't provide a lot of information as it creates class files. In fact, if the source code compiles successfully and no deprecated methods are employed, you won't see any output from the compiler. No news is good news in this case.

If you want to see more information on what the `javac` tool is doing as it compiles source code, use the `-verbose` option. The more verbose compiler describes how long it takes to complete different functions, the classes that are being loaded, and the overall time required.

The `appletviewer` Browser

The `appletviewer` tool runs Java programs that require a web browser and are presented as part of a Hypertext Markup Language (HTML) document. It takes an HTML document as a command-line argument, as in the following example:

```
appletviewer NewAuctions.html
```

If the argument is a web address instead of a reference to a file, `appletviewer` loads the HTML document at that address. For example:

Click here to view code image

```
appletviewer http://www.javaonthebrain.com
```

Figure E.1 shows an applet loaded from this page, a site developed by cartoonist and Java game programmer Karl Hörnell.

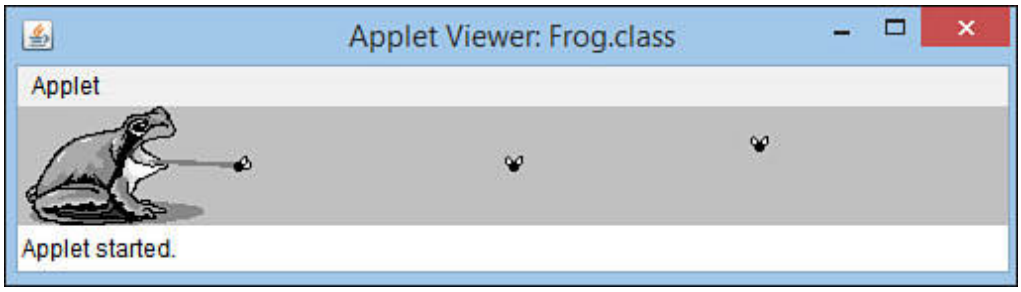


FIGURE E.1 Viewing Java web applets outside of a browser.

When `appletviewer` loads an HTML document, every applet on that document begins running in its own window. The size of these windows depends on the `height` and `width` attributes that were set in the applet's `html` tag.

Unlike a web browser, `appletviewer` cannot be used to view the HTML document itself. If you want to see how the applet is laid out in relation to the other contents of the document, you must use a Java-capable web browser.

Caution

The Java Plug-in from Oracle enables web browsers to run Java applets. The Plug-in is included in the Java Runtime Environment, a JVM for running Java programs that is installed along with the Java Development Kit. If it isn't already present on your system, you can download it from Oracle's website at www.java.com.

Using `appletviewer` is reasonably straightforward, but you might be unfamiliar with some of the menu options that are available as the viewer runs an applet.

The following menu options are available:

- The Restart and Reload options are used to restart the applet's execution. The difference between these two options is that Restart does not unload the applet before restarting it, whereas Reload does. The Reload option is equivalent to closing the applet viewer and opening it again on the same web page.
- The Start and Stop options are used to call the applet's `start()` and `stop()` methods directly.
- The Clone option creates a second copy of the same applet running in its own window.
- The Tag option displays the program's `applet` or `object` tag, along with the HTML for any `param` tags that configure the applet.

Another option on the Applet pull-down menu is Info, which calls the applet's `getAppletInfo()` and `getParameterInfo()` methods. A programmer can implement these methods to provide more information about the applet and the parameters it can handle.

The `getAppletInfo()` method returns a string that describes the applet. The `getParameterInfo()` method returns an array of string arrays that specify the name, type, and description of each parameter.

Listing E.1 contains a Java applet that demonstrates the use of these methods.

LISTING E.1 The Full Text of `AppInfo.java`

Click here to view code image

```
 1: import java.awt.*;
 2:
 3: public class AppInfo extends javax.swing.JApplet {
 4:     String name, date;
 5:     int version;
 6:
 7:     public String getAppletInfo() {
 8:         String response = "This applet demonstrates the "
 9:             + "use of the Applet's Info feature.";
10:         return response;
11:     }
```

```
12:
13:     public String[][] getParameterInfo() {
14:         String[] p1 = { "Name", "String", "Programmer's name"
};
15:         String[] p2 = { "Date", "String", "Today's date" };
16:         String[] p3 = { "Version", "int", "Version number" };
17:         String[][] response = { p1, p2, p3 };
18:         return response;
19:     }
20:
21:     public void init() {
22:         name = getParameter("Name");
23:         date = getParameter("Date");
24:         String versText = getParameter("Version");
25:         if (versText != null) {
26:             version = Integer.parseInt(versText);
27:         }
28:     }
29:
30:     public void paint(Graphics screen) {
31:         Graphics2D screen2D = (Graphics2D) screen;
32:         screen2D.drawString("Name: " + name, 5, 50);
33:         screen2D.drawString("Date: " + date, 5, 100);
34:         screen2D.drawString("Version: " + version, 5, 150);
35:     }
36: }
```

The main function of this applet is to display the value of three parameters: `Name`, `Date`, and `Version`. The `getAppletInfo()` method returns the following string:

Click here to view code image

```
This applet demonstrates the use of the Applet's Info feature.
```

The `getParameterInfo()` method is a bit more complicated if you haven't worked with multidimensional arrays. The following things are taking place:

- Line 13 defines the return type of the method as a two-dimensional array of `String` objects.
- Line 14 creates an array of `String` objects with three elements: "Name", "String", and "Programmer's name". These elements describe one of the parameters that can be defined for the AppInfo applet. They describe the name of the parameter ("Name" in this case), the type of data that the parameter will hold (a string), and a description of the

parameter ("Programmer's name"). The three-element array is stored in the `p1` object.

- Lines 15 and 16 define two more `String` arrays for the `Date` and `Version` parameters.
- Line 17 uses the `response` object to store an array that contains three string arrays: `p1`, `p2`, and `p3`.
- Line 18 uses the `response` object as the method's return value.

Listing E.2 contains a web page that can be used to load the AppInfo applet.

LISTING E.2 The Full Text of `AppInfo.html`

Click here to view code image

```
1: <applet code="AppInfo.class" height="200" width="170">
2: <param name="Name" value="Rogers Cadenhead">
3: <param name="Date" value="08/15/15">
4: <param name="Version" value="7">
5: </applet>
```

Figure E.2 shows the applet running with `appletviewer`, and Figure E.3 is a screen capture of the dialog box that opens when the viewer's Info menu option is selected.

FIGURE E.2 The AppInfo applet running in `appletviewer`.

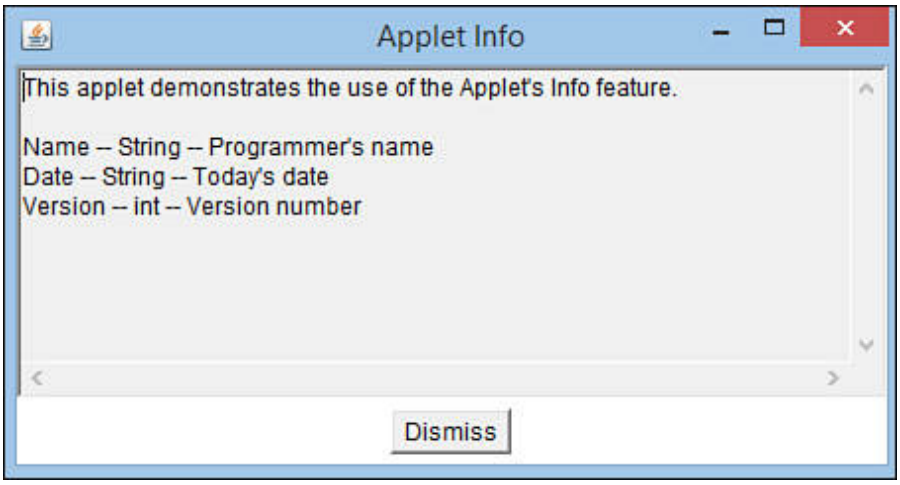


FIGURE E.3 The Info dialog box of the AppInfo applet.

These features require a browser that makes this information available to users. The kit's `appletviewer` handles this through the Info menu option, but actual browsers do not offer anything like it.

The `javadoc` Documentation Tool

The Java documentation creator, `javadoc`, takes a `.java` source code file or package name as input and generates detailed documentation in HTML format.

For `javadoc` to create full documentation for a program, a special type of comment statement must be used in the program's source code. Tutorial programs in this book use `//`, `/*`, and `*/` in source code to create *comments*—information for people who are trying to make sense of the program.

Java also has a more structured type of comment that the `javadoc` tool can read. This comment is used to describe program elements such as classes, variables, objects, and methods. It takes the following format:

Click here to view code image

```
/** A descriptive sentence or paragraph.
  * @tag1 Description of this tag.
  * @tag2 Description of this tag.
 */
```

A Java documentation comment should be placed immediately above the program element it is documenting and should succinctly explain what the program element is. For example, if the comment precedes a `class` statement, it describes the class's purpose.

In addition to the descriptive text, different items can be used to document the program element further. These items, called *tags*, are preceded by an at sign (`@`) and are followed by a space and a descriptive sentence or paragraph.

Listing E.3 contains a thoroughly documented version of the AppInfo applet called AppInfo2. The following tags are used in this program:

- `@author`—The program's author. This tag can be used only when a class is documented. It is ignored unless the `-author` option is used when `javadoc` is run.
- `@version` *`text`*—The program's version number. This also is restricted to class documentation. It requires the `-version` option when you're running `javadoc`, or the tag will be ignored.
- `@return` *`text`*—The variable or object returned by the method being documented.
- `@serial` *`text`*—A description of the data type and possible values for a variable or object that can be *serialized*—saved to disk along with the values of its variables and retrieved later.

LISTING E.3 The Full Text of `AppInfo2.java`

Click here to view code image

```
 1: import java.awt.*;
 2:
 3: /** This class displays the values of three parameters:
 4:   * Name, Date and Version.
 5:   * @author <a href="http://java21days.com/">Rogers
Cadenhead</a>
 6:   * @version 7.0
 7:  */
 8: public class AppInfo2 extends javax.swing.JApplet {
 9:     /**
10:       * @serial The programmer's name.
11:      */
12:     String name;
13:     /**
14:       * @serial The current date.
15:      */
16:     String date;
17:     /**
18:       * @serial The program's version number.
19:      */
20:     int version;
```

```
21:
22:     /**
23:       * This method describes the applet for any browsing
tool that
24:       * requests information from the program.
25:       * @return A String describing the applet.
26:      */
27:     public String getAppletInfo() {
28:         String response = "This applet demonstrates the "
29:             + "use of the Applet's Info feature.";
30:         return response;
31:     }
32:
33:     /**
34:       * This method describes the parameters that the applet
can take
35:       * for any browsing tool that requests this information.
36:       * @return An array of String[] objects for each
parameter.
37:      */
38:     public String[][] getParameterInfo() {
39:         String[] p1 = { "Name", "String", "Programmer's name"
};
40:         String[] p2 = { "Date", "String", "Today's date" };
41:         String[] p3 = { "Version", "int", "Version number" };
42:         String[][] response = { p1, p2, p3 };
43:         return response;
44:     }
45:
46:     /**
47:       * This method is called when the applet is first
initialized.
48:      */
49:     public void init() {
50:         name = getParameter("Name");
51:         date = getParameter("Date");
52:         String versText = getParameter("Version");
53:         if (versText != null) {
54:             version = Integer.parseInt(versText);
55:         }
56:     }
57:
58:     /**
59:       * This method is called when the applet's display
window is
60:       * being repainted.
61:      */
62:     public void paint(Graphics screen) {
63:         Graphics2D screen2D = (Graphics2D) screen;
64:         screen.drawString("Name: " + name, 5, 50);
65:         screen.drawString("Date: " + date, 5, 100);
```

```
66:          screen.drawString("Version: " + version, 5, 150);
67:      }
68: }
```

The following command creates HTML documentation from the source code file `AppInfo2.java`:

Click here to view code image

```
javadoc -author -version AppInfo2.java
```

The Java documentation tool creates several different web pages in the same folder as `AppInfo2.java`. These pages document the program in the same manner as Oracle's official documentation for the Java Class Library.

Tip

To see the official documentation for Java 8 and the Java Class Library, visit http://docs.oracle.com/javase/8/docs/api.

To see the documentation that `javadoc` has created for AppInfo2, load the newly created web page `index.html` on your web browser. Figure E.4 shows this page loaded with Google Chrome.

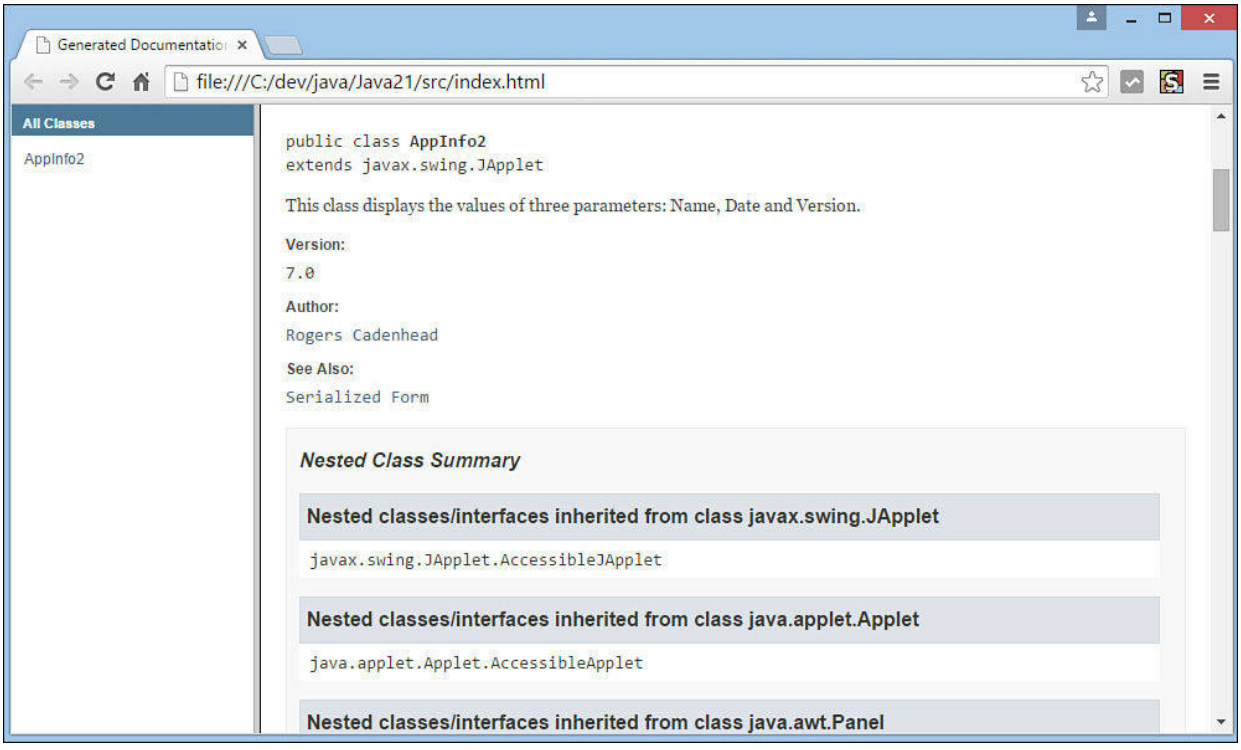


FIGURE E.4 Java documentation for the AppInfo2 program.

The `javadoc` tool produces extensively hyperlinked web pages. Navigate through the pages to see where the information in your documentation comments and tags shows up.

If you're familiar with HTML markup, you can use HTML tags such as `A`, `TT`, and `B` within your documentation comments. Line 5 of the AppInfo2 program uses an `A` tag to turn the text "Rogers Cadenhead" into a hyperlink to this book's website.

The `javadoc` tool also can be used to document an entire package by specifying the package name as a command-line argument. HTML files are created for each `.java` file in the package, along with an HTML file indexing the package.

If you want the Java documentation to be produced in a different folder than the default, use the `-d` option followed by a space and the folder name.

The following command creates Java documentation for AppInfo2 in a folder called `C:\JavaDocs\`:

Click here to view code image

```
javadoc -author -version -d C:\JavaDocs\ AppInfo2.java
```

The following list details the other tags you can use in Java documentation comments:

- `@deprecated` *`text`* provides a note that indicates that the class, method, object, or variable has been deprecated. This causes the `javac` compiler to issue a deprecation warning when the feature is used in a program that's being compiled.
- `@exception` *`class description`* is used with methods that throw exceptions. This tag documents the exception's class name and its description.
- `@param` *`name description`* is used with methods. This tag documents the name of an argument and a description of the values the argument can hold.
- `@see` *`class`* indicates the name of another class, which will be turned into a hyperlink to the Java documentation for that class. This can be used without restriction in comments.
- `@see` *`class#method`* indicates the name of a method of another class, which will be used for a hyperlink directly to the documentation

for that method. This can be used without restriction.

- `@since` *`text`* indicates a note describing when a method or feature was added to Java's class library.

The `jar` Java File Archival Tool

When you deploy a Java program, keeping track of all the class files and other files required by the program can be cumbersome.

To make this easier, the kit includes a tool called `jar` that can pack all a program's files into a Java archive—also called a JAR file. The `jar` tool also can be used to unpack the files in one of these archives.

JAR files can be compressed using the zip format or packed without using compression.

To use the tool, type the command `jar` followed by command-line options and a series of filenames, folder names, or wildcards.

The following command packs all of a folder's class and GIF image files into a single Java archive called `Animate.jar`:

Click here to view code image

```
jar cf Animate.jar *.class *.gif
```

The argument `cf` specifies two command-line options that can be used when running the `jar` program. The `c` option indicates that a Java archive file should be created, and `f` indicates that the name of the archive file will follow as one of the next arguments.

You also can add specific files to a Java archive with a command such as the following:

Click here to view code image

```
jar cf MusicLoop.jar MusicLoop.class muskratLove.mp3
shopAround.mp3
```

This creates a `MusicLoop.jar` archive containing three files: `MusicLoop.class`, `muskratLove.mp3`, and `shopAround.mp3`.

Run `jar` without any arguments to see a list of options that can be used with the tool.

One use of `jar` is to put all files necessary to run a Java applet in a single JAR file. This makes it much easier to deploy the applet on the Web.

The standard way of placing a Java applet on a web page is to use an `applet` or `object` tag to indicate the applet's primary class file. A Java-enabled browser then downloads and runs the applet. Any other classes and any other files that the applet needs are downloaded from the web server.

The problem with running applets in this way is that every file an applet requires—helper classes, images, audio files, text files, or anything else—requires a separate connection from a web browser to the server containing the file. This can significantly increase the amount of time it takes to download an applet and everything it needs to run.

If you can reduce the number of files the browser has to load from the server by putting many files into one Java archive, a web browser can download and run your applet more quickly. If the files in a Java archive are compressed, it loads even more quickly.

After you create a Java archive, the `archive` attribute is used with the `applet` tag to show where the archive can be found. You can use Java archives with an applet with tags such as the following:

Click here to view code image

```
<applet code="MusicLoop.class" archive="MusicLoop.jar" width="45"
height="42">
</applet>
```

This tag specifies that an archive called `MusicLoop.jar` contains files used by the applet. Browsers and browsing tools that support JAR files will look inside the archive for files that are needed as the applet runs.

Caution

Although a Java archive can contain class files, the `archive` attribute does not remove the need for the `code` attribute. A browser still needs to know the name of the applet's main class file to load it.

When you use an `object` tag to display an applet that uses a JAR file, the applet's archive file is specified as a parameter using the `param` tag. The tag should have the `name` attribute "archive" and a `value` attribute with the name of the archive file.

The following example is a rewrite of the preceding example to use `object` instead of `applet`:

Click here to view code image

```
<object code="MusicLoop.class" width="45" height="42">
    <param name="archive" value="MusicLoop.jar">
</object>
```

The `jdb` Debugger

`jdb`, the Java debugger, is a sophisticated tool that helps you find and fix bugs in Java programs. You also can use it to better understand what is taking place behind the scenes in the JVM as a program is running. It has a large number of features, including some that might be beyond the expertise of a Java programmer who is new to the language.

You don't need to use the debugger to debug Java programs. This is fairly obvious, especially if you've been creating your own Java programs as you read this book. After the Java compiler generates an error, the most common response is to load the source code into an editor, find the line cited in the error message, and try to spot the problem. You repeat this dreaded compile-curse-find-fix cycle until the program compiles without complaint.

After using this debugging method for a while, you might think that the debugger is unnecessary to the programming process because it's such a complicated tool to master. This reasoning makes sense when you're fixing problems that cause compiler errors. Many of these problems are simple things such as a misplaced semicolon, unmatched `{` and `}` braces, or the use of the wrong type of data as a method argument. However, when you start looking for logic errors—more subtle bugs that don't stop the program from compiling and running—a debugger is an invaluable tool.

The Java debugger has two features that are useful when you're searching for a bug that can't be found by other means: single-step execution and breakpoints. Single-step execution pauses a Java program after every line of code is executed. Breakpoints are points where execution of the program pauses. Using the Java debugger, these breakpoints can be triggered by specific lines of code, method calls, or caught exceptions.

The Java debugger works by running a program using a version of the JVM over which it has complete control.

Before you use the Java debugger with a program, you compile the program with the `-g` option, which causes extra information to be included in the class file. This information greatly aids in debugging. Also, you shouldn't use the -

`O` option, because its optimization techniques might produce a class file that does not directly correspond with the program's source code.

Debugging Applications

If you're debugging an application, you can run the `jdb` tool with a Java class as an argument. This is shown in the following:

```
jdb WriteBytes
```

This example runs the debugger with `WriteBytes.class`, an application that's available from the book's website at www.java21days.com. Visit the site, select the Appendix E page, and then save the files `WriteBytes.class` and `WriteBytes.java` in the same folder from which you run the debugger.

The WriteBytes application writes a series of bytes to disk to produce the file `pic.gif`.

The debugger loads this program but does not begin running it, displaying the following output:

```
Initializing jdb...
>
```

You control the debugger by typing commands at the > prompt.

To set a breakpoint in a program, you use the `stop in` or `stop at` commands. The `stop in` command sets a breakpoint at the first line of a specific method in a class. You specify the class and method name as an argument to the command, as in the following example:

```
stop in SellItem.SetPrice
```

This command sets a breakpoint at the first line of the `SetPrice()` method. Note that no arguments or parentheses are needed after the method name.

The `stop at` command sets a breakpoint at a specific line number within a class. You specify the class and number as an argument to the command, as in the following example:

```
stop at WriteBytes:14
```

If you're trying this with the `WriteBytes` class, you see the following output after entering this command:

Click here to view code image

```
Deferring breakpoint WriteBytes:14
It will be set after the class is loaded.
```

You can set as many breakpoints as you want within a class. To see the breakpoints that are currently set, use the `clear` command without any arguments. The `clear` command lists all current breakpoints by line number rather than method name, even if they were set using the `stop in` command.

By using `clear` with a class name and line number as an argument, you can remove a breakpoint. If the hypothetical `SellItem.SetPrice` method were located at line 215 of `SellItem`, you could clear this breakpoint with the following command:

```
clear SellItem:215
```

Within the debugger, you can begin executing a program with the `run` command. The following output shows what the debugger displays after you begin running the `WriteBytes` class:

Click here to view code image

```
run WriteBytes
VM Started: Set deferred breakpoint WriteBytes:14

Breakpoint hit: "thread=main", WriteBytes.main(), line=14 bci=413
14                for (int i = 0; i < data.length; i++)
```

After you have reached a breakpoint in the `WriteBytes` class, experiment with the following commands:

- `list`—At the point where execution stopped, this command displays the source code of the line and several lines around it. This requires access to the `.java` file of the class where the breakpoint has been hit, so you must have `WriteBytes.java` in either the current folder or one of the folders in your `Classpath`.
- `locals`—This command lists the values for local variables that are currently in use or will soon be defined.
- `print` *`text`*—This command displays the value of the variable, object, or array element specified by *`text`*.
- `step`—This command executes the next line and stops again.
- `cont`—This command continues running the program at the point it was halted.
- `!!`—This command repeats the previous debugger command.

After trying out these commands within the application, you can resume running the program by clearing the breakpoint and using the `cont` command. Use the `exit` command to end the debugging session.

The WriteBytes application creates a file called `pic.gif`. You can verify that this file ran successfully by loading it with a web browser or image-editing software. You'll see a small letter J in black and white.

After you have finished debugging a program and you're satisfied that it works correctly, recompile it without the `-g` option.

Debugging Applets

You can't debug an applet by loading it using the `jdb` tool. Instead, use the `-debug` option of `appletviewer`, as in the following example:

Click here to view code image

```
appletviewer -debug AppInfo.html
```

This loads the Java debugger, and when you use a command such as `run`, `appletviewer` begins running also. Try this example to see how these tools interact.

Before you use the `run` command to execute the applet, set a breakpoint in the program at the first line of the `getAppletInfo` method. Use the following command:

Click here to view code image

```
stop in AppInfo.getAppletInfo
```

After you begin running the applet, the breakpoint won't be hit until you cause the `getAppletInfo()` method to be called. You do so by selecting Applet, Info from `appletviewer`'s menu.

Advanced Debugging Commands

With the features you have learned about so far, you can use the debugger to stop execution of a program and learn more about what's taking place. This might be sufficient for many of your debugging tasks, but the debugger also offers many other commands. These include the following:

- `up` moves up the stack frame so that you can use `locals` and `print` to examine the program at the point before the current method was called.

- `down` moves down the stack frame so that you can examine the program after the method call.

A Java program often has places where a chain of methods is called. One method calls another method, which calls another method, and so on. At each point where a method is being called, Java keeps track of all the objects and variables within that scope by grouping them. This grouping is called a *stack*, as if you were stacking these objects like a deck of cards. The various stacks in existence as a program runs are called the stack frame.

By using `up` and `down` along with commands such as `locals`, you can better understand how the code that calls a method interacts with that method.

You also can use the following commands within a debugging session:

- `classes` lists the classes currently loaded into memory.
- `methods` lists the methods of a class.
- `memory` shows the total amount of memory and the amount that isn't currently in use.
- `threads` lists the threads that are executing.

The `threads` command numbers all the threads. This enables you to use the `suspend` command followed by a number to pause that thread, as in `suspend 1`. You can resume a thread by using the `resume` command followed by the thread's number.

Another convenient way to set a breakpoint in a Java program is to use the `catch` *`text`* command, which pauses execution when the `Exception` class named by *`text`* is caught.

You also can cause an exception to be ignored by using the `ignore` *`text`* command with the `Exception` class named by *`text`*.

Using System Properties

One handy feature of the kit is that the command-line option `-D` can modify the performance of the Java Class Library.

If you have used other programming languages before learning Java, you might be familiar with environment variables, which provide information about the operating system in which a program is running. An example is the `Classpath` setting, which indicates the folders where the JVM should look for a class file.

Because different operating systems have different names for their environment variables, a Java program cannot read them directly. Instead, Java includes a number of different system properties that are available on any platform with a Java implementation.

Some properties are used only to get information. The following system properties are among those that should be available on any Java implementation:

- `java.version` is the version number of the JVM.
- `java.vendor` is a string identifying the vendor associated with the JVM.
- `os.name` is the operating system in use.
- `os.version` is the version number of that operating system.

Other properties can affect how the Java Class Library performs when being used inside a Java program. An example is the `java.io.tmpdir` property, which defines the folder that Java's input and output classes use as a temporary workspace.

You can set a property at the command line by using the `-D` option followed by the property name, an equal sign, and the property's new value, as in this command:

Click here to view code image

```
java -Duser.timezone=Asia/Jakarta Auctioneer
```

The use of the system property in this example sets the default time zone to Asia/Jakarta before running the `Auctioneer` class. This affects any `Date` objects in a Java program that do not set their own zone.

These property changes are not permanent; they apply only to that particular execution of the class and any classes it uses.

Tip

In the `java.util` package, the `TimeZone` class includes a class method called `getProperties()` that returns a string array containing all the time zone identifiers that Java supports.

The following code displays these identifiers:

Click here to view code image

```
String[] ids = java.util.TimeZone.getAvailableIDs();
for (int i = 0; i < ids.length; i++) {
    System.out.println(ids[i]);
}
```

You also can create your own properties and read them using the `getProperty()` method of the `System` class, which is part of the `java.lang` package.

Listing E.4 contains the source code of a simple program that displays the value of a user-created property.

LISTING E.4 The Full Text of `ItemProp.java`

Click here to view code image

```
1: class ItemProp {
2:     public static void main(String[] arguments) {
3:         String n = System.getProperty("item.name");
4:         System.out.println("The item is named " + n);
5:     }
6: }
```

If you run this program without setting the `item.name` property on the command line, the output is the following:

```
The item is named null
```

You can set the `item.name` property using the `-D` option, as in this command:

Click here to view code image

```
java -Ditem.name="Microsoft Bob" ItemProp
```

The output is the following:

Click here to view code image

```
The item is named Microsoft Bob
```

The `-D` option is used with the JVM. To use it with `appletviewer` as well, all you have to do differently is precede the `-D` with `-J`. The following command shows how this can be done:

Click here to view code image

```
appletviewer -J-Dtimezone=Asia/Jakarta AppInfo.html
```

This example causes `appletviewer` to use the default time zone Asia/Jakarta with all applets on the web page `AppInfo.html`.

The `keytool` and `jarsigner` Code Signing Tools

On Day 14, "Developing Swing Applications," the process of packaging an application into a JAR file that can be run from a web page is described. This requires that the JAR be signed with a digital certificate. For testing purposes, the `keystore` and `jarsigner` tools in the KIT can be used to create a key and use it to digitally sign a JAR file.

The first step is to use `keytool` to create a key and assign it an alias and password:

Click here to view code image

```
keytool -genkey -alias examplekey -keypass swordfish
```

The `-genkey` argument generates a new key, which in this example is named "examplekey" and has the password "swordfish". If this is the first time `keytool` has been used, you're prompted for a password that protects access to the key database, which is called a *keystore*.

After a key has been placed in the keystore, it can be used with the `jarsigner` tool to sign an archive file. This tool requires the keystore and key passwords and the key's alias. Here's how the `Animate.jar` archive could be signed with the `examplekey` key:

Click here to view code image

```
jarsigner -storepass bazinga -keypass swordfish Animate.jar
examplekey
```

The keystore password in this example is "bazinga". The security certificate used to sign the archive will last 90 days and will be described as an

“untrusted source” when an application is run through Java Web Start or when an applet is run in a web browser.

Because running a Java program in a browser from an untrusted source violates Java’s default security policy, the only way to run a self-signed JAR file is to add an exception to the Java Control Panel. Run the control panel, which on Windows is part of the Windows Control Panel. Choose the Security tab to bring it to the front; then click Edit Site List to add the domain name of the source of the applet (such as `http://cadenhead.org` if you wanted to run the PageData Java Web Start application from the book’s website at www.java21days.com).

Index

Symbols

$ (dollar sign) in variable names, 41
& (ampersand), AND operators, 57
-> (arrow operator), 460
* (asterisk) multiplication operator, 52
*= (asterisk equal) assignment operator, 54
^ (caret) XOR operator, 58
!! command (jdb), 654
/* comment notation, 46
/** comment notation, 47
// comment notation, 46
{} (curly braces), 38
  block statements, 103
<> (diamond operator), 247
&& (double ampersand), AND operators, 57
|| (double pipe character), OR operators, 58
== (equal) comparison operator, 57, 88
= (equal sign) assignment operator, 40, 43, 54-55
! (exclamation point) NOT operator, 58
/ (forward slash) division operator, 52
/= (forward slash equal) assignment operator, 54
> (greater than) comparison operator, 57
>= (greater than or equal to) comparison operator, 57
< (less than) comparison operator, 57
<= (less than or equal to) comparison operator, 57
-= (minus equal) assignment operator, 54
- (minus sign)
  decrement operator (--), 55-56
  negative numbers, 48
  subtraction operator, 52

!= (not equal) comparison operator, 57, 88
() (parentheses)
  arguments, 68
  grouping expressions, 59-60
% (percent sign) modulus operator, 52
. (period)
  accessing methods and variables, 59
  dot notation, 73
| (pipe character) OR operators, 58
+= (plus equal) assignment operator, 54
+ (plus sign)
  addition operator, 52
  concatenation operator, 82
  increment operator (++), 55-56
  string concatenation, 60-61
? (question mark) in SQL statements, 514
“” (quotation marks) in arguments, 137
; (semicolon) statement termination character, 38
/ (slash character), XML tags, 401
[] (square brackets), arrays, 59, 96
_ (underscore) in large number literals, 48
2D graphics. *See* Java2D

A

absolute component placement, 334
abstract classes, 169
abstract methods, 169, 187
abstract modifier, 158, 169
Abstract Windowing Toolkit (AWT). *See* java.awt package
accept() method, 479
access control, 159
  accessor methods, 164
  comparison of types, 163

default access, 159, 175
inheritance, 163
interfaces, 179
packages, 175
private access, 159-161
protected access, 162
public access, 161, 175
Access databases, Java DB versus, 523
accessing
array elements, 98-99, 233
class methods, 165
class variables, 75, 165
databases, 505
instance variables, 72-73
accessor methods, 164, 187
action events, 340, 345-346
ActionListener interface, 330, 340, 345, 460
actionPerformed() method, 330, 342, 345
activities (Android apps), 585
acyclic gradients, 378
adapter classes, 357-359, 456-457
addActionListener() method, 330, 341, 345
addAttribute() method, 533
addFocusListener() method, 341
addHandler() method, 560
adding
Apache XML-RPC to NetBeans, 555
child nodes to parent nodes, 533
classes to packages, 175
components
to containers, 256, 262-263
to panels, 325
to toolbars, 298

JavaDB library to projects, 512
separators to menus, 304
stack elements, 239
white space to XML documents, 540-542
XOM to NetBeans, 532
addItemListener() method, 341
addItem() method, 275
addition operator, 52
add() method
array lists, 233-234
border layouts, 323
card layouts, 326
check boxes/radio buttons, 273
containers, 262
menus, 304
addMouseListener() method, 341
addMouseMotionListener() method, 341
addSeparator() method, 304
addTab() method, 307
addTextListener() method, 341
addWindowListener() method, 341
adjustment events, 340
AdjustmentListener event listener, 340
Advogato, 554
afterLast() method, 521
aligning
components
border layouts, 322-324
box layouts, 317-319
card layouts, 325-333
flow layouts, 315-317
grid layouts, 320-321
panels, 325

labels, 267
AllCapsDemo.java, 442
allocate() method, 485
allocating memory, 71
all-permissions tag, 406
Alphabet.java, 316
ampersand (&), AND operators, 57
AND operators, 57
Android, 569
apps, 569
closing projects in Android Studio, 579
configuring manifest files in Android Studio, 581
creating projects in Android Studio, 572-574, 579
designing GUI in Android Studio, 581-584
installing and configuring Android Studio, 571-572
organizing projects in Android Studio, 574-575
preparing resources in Android Studio, 579-580
running, 577-578, 589-591
strings in, 575-577
troubleshooting, 578, 610-614
writing in Android Studio, 575-577
writing Java code in Android Studio, 584-591
history of, 570-571
versions of, 592
android.content package, 587
Android Developer site, 570
AndroidManifest.xml, 579-581
Android SDK Manager, 592
installing HAXM, 611-612
Android Software Development Kit (SDK), 570
Android Studio, 13, 570
closing projects, 579
configuring manifest files, 581

creating projects, 572-574, 579
designing GUI, 581-584
installing and configuring, 571-572
organizing projects, 574-575
preparing resources, 579-580
running apps in, 577-578, 589-591
troubleshooting, 578, 610
checking BIOS settings, 614
installing HAXM, 611-613
writing apps in, 575-577
writing Java code in, 584-591
android.support.v7.app package, 586
angle brackets (<>), diamond operator, 247
anonymous inner classes, 454-459, 466
antialiasing, 372
Apache Derby, 503
Apache Project class libraries, 279
Apache XML-RPC
clients, 556-559
data types supported, 565
installing, 554-556
servers, 559-564
AppCompatActivity class, 586
appendChild() method, 533
append() method, 269
AppInfo.html, 645
AppInfo.java, 644
AppInfo2.java, 647
applet-desc tag, 413
applets, 391
converting to applications, 413
debugging, 655
Java Web Start applications versus, 393

linking, URL objects, 471
appletviewer browser, 642-646
application-desc tag, 403
applications (Java), 136
arguments
handling, 138-139
passing to, 137
Buttons.java, 263
converting applets to, 413
creating, 135-136
debugging, 653-655
deployment
configuring web servers for Java Web Start, 405
creating JNLP files, 396-404
description tag, 406
icon tag, 406-407
Java Web Start, 392-395
JNLP security, 405-406
digital signatures, 404
multitasking. *See* threads
performance improvements, 407-412
running, 602-603, 639
server applications
designing, 480-482
testing, 482-483
splash screens, 407
Storefront, 181-187
Swing
creating interface, 257-259
developing framework, 260-261
threaded
example, 213-217
writing, 211-213

apps (Android), 569
closing projects in Android Studio, 579
configuring manifest files in Android Studio, 581
creating projects in Android Studio, 572-574, 579
designing GUI in Android Studio, 581-584
installing and configuring Android Studio, 571-572
organizing projects in Android Studio, 574-575
preparing resources in Android Studio, 579-580
running, 577-578, 589-591
strings in, 575-577
troubleshooting, 578, 610
checking BIOS settings, 614
installing HAXM, 611-613
writing in Android Studio, 575-577
writing Java code in Android Studio, 584-591
Arc2D.Float class, 382-383
archiving files, 650-652
arcs, drawing with Arc2D.Float class, 382-383
arguments
command-line arguments, 638
setting, 109
storing, 111
troubleshooting, 139
creating objects, 68
grouping, 137
handling in applications, 138-139
passing
to applications, 137
to methods, 132-134
quotation marks in, 137
running programs, 623
in XML-RPC, 553
argument tag, 403

arithmetic, string, 60-61
arithmetic operators, 52-54
ArrayCopier.java, 117
array data type (XML-RPC), 550
ArrayIndexOutofBounds exception, 194
ArrayList class, 227, 233-235
ArrayList() constructor, 556
ArrayList object, 556
array lists
  accessing elements, 233
  creating, 233
  looping through, 235-238
arrays, 96, 226. *See also* loops
  Boolean arrays, data structures versus, 251
  boundaries, 99
  of bytes, casting objects to, 565
  of command-line arguments, 111
  compilation errors, 99
  elements
    accessing, 98-99
    changing, 99-102
    data types, 98
    in grids, 102
  limitations, 226
  multidimensional, 102
  objects, creating, 97-98
  references, 99
  subscripts, 98-99
  troubleshooting, 99
  variables, declaring, 96-97
arrow operator (->), 460
ASCII character set, 437, 487
assigning values to variables, 40, 43, 62

assignment operators, 54-55
  equal sign (=), 40, 43
associating
  components with event listeners, 341-342
  filters, 421
  .java files with text editor, 630
  MIME types, 405
asterisk (*) multiplication operator, 52
asterisk equal (*=) assignment operator, 54
Atom, 528, 546
attributes, 17-18
  in class hierarchies, 29
  creating, 533
  defining, 17-18
  XML tags, 401, 527
Authenticator.java, 269
Authenticator2.java, 272
author contact information, 608
@author tag (javadoc), 647
autoboxing, 87, 247
AUTOEXEC.BAT, 627-628
Averager.java, 138
AWT (Abstract Windowing Toolkit). *See* java.awt package

B

background color, setting, 377
base-2 numbering system, 48
base-8 numbering system, 48
base-16 numbering system, 49
base64 data type (XML-RPC), 550
BasicStroke class, 380-381
beforeFirst() method, 521
behavior, 18-19

shared, 31-32
binary numbers, 48
BIOS settings, checking, 614
bits, 227-232
BitSet class, 227-232
bitwise operators, 59
block statements, 38, 103
scope, 121
try and catch, 196-199
finally clause, 199-202
book website, 607
Boolean arrays, data structures versus, 251
boolean data type, 42
casting, 83
XML-RPC, 550
Boolean literals, 49
Border.java, 323
BorderLayout class, 298, 322
BorderLayout() constructor, 322
border layout manager, 322-324
boundaries, arrays, 99
Box.java, 141
Box2.java, 146
BoxLayout class, 317
box layout manager, 317-319
braces. *See* curly braces ({})
brackets. *See* square brackets ([])
breaking loops, 119
break keyword, 107, 119-120
breakpoints, 652
deleting, 654
setting, 653-655
browser (appletviewer), 642-646

BufferConverter.java, 490
BufferDemo.java, 429
buffered character streams
  reading, 438
  writing, 440
BufferedInputStream class, 427
BufferedInputStream() constructor, 428
BufferedOutputStream class, 427
BufferedOutputStream() constructor, 428
BufferedReader class, 438
BufferedReader() constructor, 438
buffered streams, 427-431
BufferedWriter class, 440
BufferedWriter() constructor, 440
buffers, 427, 484-486
  byte buffers, 486
  channels, 488-491
  character sets, 487-488
  nonblocking I/O network connections, 492-499
Builder class, 536
Builder() constructor, 536
build() method, 536
built-in fonts, 371
Bunch.java, 320
ButtonFrame.java, 262
ButtonGroup object, 273
buttons, 255, 261
  differentiating in mouse events, 362
  event handling
    action events, 345-346
    item events, 349-351
  fonts, changing, 281
  ImageButton widgets (Android), 582-584

Buttons.java, 263
ByteBuffer object, 489
byte buffers, 486-488
bytecode, 11, 639
byte data type, 42, 83
byte filters, 427
ByteReader.java, 424
bytes
  arrays of, casting objects to, 565
  multiple, writing, 425
  unsigned, 434
byte streams, 419-422
  file input streams, 422-425
  file output streams, 425-427
ByteWriter.java, 426

C

C programs, reading, 444
C++, Java versus, 11
Calculator.java, 347
calling
  constructors, 144, 151
    from another constructor, 145-146
  methods, 18, 75-77
    class methods, 80
    nesting calls, 78-79
    in superclasses, 150
CardLayout class, 326
card layout manager, 325-333
cards, 326
caret (^), XOR operator, 58
case keyword, 107
case-sensitivity of Java, 41

casting. *See also* converting
definition of, 83
destinations, 83
explicit casts, 84
objects, 82-85, 565
primitive types, 82-84
sources, 83
catching exceptions, 194-196
try and catch blocks, 196-199
finally clause, 199-202
CD command (MS-DOS), 621-622
certificate authorities, 404
chaining methods, 655
changing. *See also* modifying
array elements, 99-102
button fonts, 281
channel() method, 495
channels, 488-499
character buffers, converting to byte buffers, 487-488
character encodings, 541
character literals, 49-50
character sets
buffers, 487-488
Unicode, 40
escape codes, 49-50
character streams, 419-420, 437
reading text files, 437-440
writing text files, 440-441
charAt() method, 77, 91
char data type, 42
casting, 83
int data type versus, 445
Charset class, 487

CharsetDecoder class, 487
CharsetEncoder class, 487
charWidth() method, 373
check boxes, 255, 272-274
  event handling
    action events, 345-346
    item events, 349-351
  nonexclusive, 273
checked exceptions, 204, 195
child nodes, adding to parent nodes, 533
.class extensions, 639
classes, 11, 14-16. *See also* packages
  abstract classes, 169
  adapter classes, 357-359, 456-457
  adding to packages, 175
  AppCompatActivity, 586
  Arc2D.Float, 382-383
  ArrayList, 227, 233-235
  attributes, 17-18
  BasicStroke, 380-381
  behavior, 18-19
  BitSet, 227-232
  BorderLayout, 298, 322
  BoxLayout, 317
  BufferedInputStream, 427
  BufferedOutputStream, 427
  BufferedReader, 438
  BufferedWriter, 440
  Builder, 536
  CardLayout, 326
  Charset, 487
  CharsetDecoder, 487
  CharsetEncoder, 487

Color, 32, 375
ColorSpace, 375
compiling, 21, 601
Component, 335
ConfirmDialog, 286-288
constants, 43-44
Container, 256
creating, 19-22, 600-602
Cursor, 461
DataInputStream, 422
DataOutputStream, 422
defining, 126
Dictionary, 227
Dimension, 258
DriverManager, 509
Ellipse2D.Float, 382
Error, 194
Exception, 194-195, 198
exception classes, 208
File, 441
FileInputStream, 422
FileOutputStream, 422
FileReader, 437
Files, 442
FileSystems, 441
FileWriter, 440, 445
FilterInputStream, 427
FilterOutputStream, 427
final classes, 167-169
FlowLayout, 314-315
FocusAdapter, 358, 457
FontMetrics, 373
Graphics, 368

Graphics2D, 368
  coordinate system, 369-370
  creating drawing surface, 368-369
GridLayout, 320
grouping, 32
HashMap, 227, 241-246
helper classes, 136, 450
hierarchies, 26-27
  creating, 27-29
  methods in, 30
HttpUrlConnection, 474
identifying, 170
importing, 171-173
InetAddress, 493
InetSocketAddress, 493
inheritance, 176
inner classes, 359-362, 388, 450-453
  anonymous inner classes, 454-459, 466
  creating, 450
  scope, 450
InputDialog, 286-289
InputStream, 422
InputStreamReader, 437
Insets, 334
instances of, creating, 16
Integer, 86
Intent, 587
interfaces versus, 176. *See also* interfaces
IOException, 195
Java Class Library, 16, 159, 225, 278-281
JButton, 16, 261
JCheckBox, 272
JComboBox, 274-275

JComponent, 256, 264
JFrame, 257
JLabel, 267
JList, 276
JMenu, 303
JMenuBar, 303
JMenuItem, 303
JOptionPane, 286
JPanel, 262, 325, 368
JPasswordField, 268
JProgressBar, 300
JRadioButton, 272
JScrollBar, 297
JScrollPane, 271, 296
JSlider, 294
JTabbedPane, 307
JTextComponent, 268
JTextField, 268
JToggleButton, 272
JToolBar, 297
Key, 388
KeyAdapter, 358, 457
libraries. *See* libraries
Line2D.Float, 381
listeners. *See* listeners
loading, 508
main classes, designating, 136
Math, 79, 280
MessageDialog, 286, 289
methods, 18
MouseAdapter, 358, 457
MouseMotionAdapter, 358
name conflicts, 172-173

NamedPoint, 151
Node, 533
Object, 26
objects, casting, 84-85
object wrappers, 86
of objects, determining, 89-90
OptionDialog, 286, 290-291
organizing, 25, 158, 170
  creating hierarchies, 27-29
  inheritance, 25-31
  interfaces, 31-32
  packages, 32, 45
OutputStream, 422
OutputStreamWriter, 440
PreparedStatement, 514
PrintStream, 79
programs versus, 125
PropertyHandlerMapping, 560
protecting, 170
Reader, 437
Rectangle2D.Float, 382
RenderingHint.Key, 388
R.java, 586
RuntimeException, 195
SelectionKey, 494
Serializer, 540
ServerSocket, 479
Socket, 475
SocketChannel, 493
SocketImpl, 480
Stack, 227, 238-239
String, 79, 201
StringTokenizer, 69

subclasses, 25-26
superclasses, 25
indicating, 126
modifying, 27
SwingWorker, 407-413
System, 79, 657
class methods, 134
in variable (input stream), 431
Text, 533
Thread, 191, 211
Throwable, 194-195
TimeZone, 657
UIManager, 259
URL, 471
variables, 126-127
Vector, 235
WebServer, 559
WindowAdapter, 358, 456
wrapper classes, 134
Writer, 437
XmlRpcClient, 556
XmlRpcServer, 559
classes command (jdb), 656
class files, specifying, 640
class keyword, 126, 450
class methods, 19, 79-80, 134-135
accessing, 165
calling, 80
defining, 134
Class not found error, 633-635
CLASSPATH variable (MS-DOS)
Windows 7-10, 633-635
Windows 98/Me, 635-636

class types, 43
class variables, 18, 39, 72, 127
  accessing, 165
  defining, 74
  initial values, 40
  instance variables versus, 33, 74
  troubleshooting, 75
  values, accessing/modifying, 75
clear command (jdb), 654
clear() method
  array lists, 235
  hash maps, 242
clients (XML-RPC), 556-559
client-side sockets
  closing, 476
  nonblocking clients, 493-499
  opening, 475
Clone command (appletviewer), 644
close() method
  buffered character streams, 441
  character streams, 440
  client-side sockets, 476
  data source connections, 512
  data streams, 434
  file output streams, 425
  streams, 420-421
closePath() method, 384
closing
  data source connections, 512
  frames, 259-260
  projects in Android Studio, 579
  socket connections, 476
closing tags (XML), 401, 527

ClosureMayhem.java, 464
closures, 449, 460-466
codebase attribute, 402
CodeKeeper.java, 236
CodeKeeper2.java, 248
code listings. *See* listings
code signing, 658-659
color, 375
  background colors, 377
  Color objects, creating, 376
  dithering, 375
  drawing colors, setting, 376-377
  finding current color, 377
  sRGB color system, 375
  XYZ color system, 375
Color class, 32, 375
Color objects, creating, 376
ColorSpace class, 375
color spaces, 375
combining layout managers, 324-325
combo boxes, 274-276
  action events, 345-346
  item events, 349-351
ComicBooks.java, 243
ComicBox.java, 452
command line, 12, 638-639
command-line arguments
  setting, 109
  storing, 111
  troubleshooting, 139
command-line interfaces, 619-621
command-line tools, javac, 631
commands. *See also* keywords; statements

import, 32-34
java -version, 624
jdb (debugger)
!!, 654
classes, 656
clear, 654
cont, 654
down, 655
exit, 655
ignore, 656
list, 654
locals, 654
memory, 656
methods, 656
print, 654
run, 654
step, 654
stop at, 653
stop in, 653
suspend, 656
threads, 656
up, 655
menu commands, appletviewer browser, 643-644
MS-DOS
CD, 621-622
CLASSPATH variable, 633-636
MD, 622
PATH variable, 625-628
comments, 46
notation, 46-47
in source code, 646
Comparable interface, 32
comparing

objects, 87-89
strings, 88-89
comparison operators, 56-57, 88
compiler errors, 210
about generics, 251
for arrays, 99
runtime errors versus, 247
compilers, 21, 641-642
compiling
classes, 21, 601
files, 641-642
Java programs in Windows, 631-632
troubleshooting, 601, 633
Component class, 335
components. *See also names of specific components*
associating with event listeners, 341-342
Swing, 256, 264
absolute placement, 334
adding to containers, 256, 262-263
adding to panels, 325
AWT components versus, 256
check boxes, 272-274
combo boxes, 274-276
creating, 256, 261-262
dialog boxes, 286-293
disabled, 264
drop-down lists, 274-276
hiding, 264
image icons, 265-267
labels, 267
layout managers. *See* layout managers
lists, 276-278
menus, 303-307

progress bars, 300-303
radio buttons, 272-274
resizing, 264
scrolling panes, 271-272, 296-297
sliders, 294-296
tabbed panes, 307-310
text areas, 269-271
text fields, 268
toolbars, 297-300
windows, frames, 257
concatenating strings, 60-61
concatenation operator (+), 82
conditional operator. *See* ternary operator
conditionals
if, 104-106
switch, 105-111, 121
ternary operator, 112
configureBlocking() method, 493
configuring
Android Studio, 571-572
Java Development Kit (JDK), 619
command-line interface, 619-621
creating folders, 622-623
opening folders, 621-622
running programs, 623
setting CLASSPATH variable, 633-636
setting PATH variable, 624-628
manifest files in Android Studio, 581
scrollbars, 271
web servers for Java Web Start applications, 405
confirm dialog boxes, 287-288
ConfirmDialog class, 286-288
conflicts, name

classes, 172-173
reducing, 170
connecting
to databases, 505-510
troubleshooting, 514
viewing connection information, 510
to Internet. *See* networking
connect() method, 493
consistency checking (exceptions), 195-196
ConsoleInput.java, 432
console. *See* command line
console input streams, 431-432
constant variables. *See* final variables
constants, 43
declaring, 44
enumerations, 249-250
naming, 44
constructors, 69, 144-145
ArrayList(), 556
BorderLayout(), 322
BufferedInputStream(), 428
BufferedOutputStream(), 428
BufferedReader(), 438
BufferedWriter(), 440
Builder(), 536
calling, 144-146, 151
DataInputStream(), 433
DataOutputStream(), 433
definition of, 71
Dimension(), 258
exception classes, 208
FileInputStream(), 422
FileOutputStream(), 425

FileReader(), 437
FileWriter(), 440
FlowLayout(), 315
GridLayout(), 320
Intent(), 587
JCheckBox(), 272
JComboBox(), 275
JFrame(), 257
JList(), 276
JMenuBar(), 305
JMenuItem(), 303
JProgressBar(), 301
JScrollPane(), 271, 296
JSlider(), 294
JTabbedPane(), 307
JTextArea(), 269
JTextField(), 268
JToolBar(), 298
naming, 144
overloading, 146-147
overriding, 150-152
Serializer(), 541
URL(), 471
WebServer(), 559
Container class, 256
containers, 255
absolute component placement, 334
adding components to, 256, 262-263
cards, 326
layout managers. *See* layout managers
menus, creating, 304
panels, 262, 325-326
contains() method, 235

containsKey() method, 242
containsValue() method, 242
cont command (jdb), 654
continue keyword, 119-120
controlling access. *See* access control
converting. *See also* casting
  applets to applications, 413
  character and byte buffers, 487-488
  primitive types and objects, 86-87
  source code, 641
  strings to numbers, 86
coordinate systems
  Java2D, 369-370
  user versus device coordinate spaces, 378
Cover Pages, 530
createFont() method, 371
createStatement() method, 510, 522
creating. *See also* constructors
  array lists, 233
  array objects, 97-98
  attributes, 533
  buffered input streams, 428
  buffered output streams, 428
  character sets, 487
  classes, 19-22, 600-602
  components, Swing, 256, 261-262
  confirm dialog boxes, 287-288
  database tables, 517
  data input streams, 433
  data output streams, 433
  drawing surfaces, 368-369
  exceptions, 207
  file input streams, 422

File objects, 441
file output streams, 425
folders in MS-DOS, 622-623
frames, 368
frameworks (GUI), 260-261
hash maps, 241
inner classes, 450
input dialog boxes, 288-289
input streams, 420
instances, 16
intents, 587
interfaces, 178-179
interfaces (GUI), 257-259
Java applications, 135-136, 629-631
JNLP files, 396-404
labels, 267
layout managers, 314
menu containers, 304
message dialog boxes, 289
methods, overloaded, 140-143
objects, 68
  arguments, 68
  with closures, 460-465
  Color, 376
  with constructors, 71
  Document, 533
  Element, 533
  Font, 370-372
  GeneralPath, 384
  ImageIcon, 265
  with new operator, 68-70
  Serializer, 540
  String, 21

StringTokenizer objects, 69-70
URL, 471
online storefronts, 181-187
option dialog boxes, 290-291
output streams, 420
overridden methods, 148-149
packages, 640
panels, 325, 368
Path objects, 442
projects, 19
Android Studio, 572-574, 579
NetBeans, 598-600
scrolling panes, 296
Selector objects, 493
server sockets, 479
source files, 629
stacks, 238
system properties, 657
threads, 212
variables, 39-40, 44-45
XML documents, 532-535
XML-RPC servers, 559-564
curly braces ({}), 38
block statements, 103
current objects, referring to, 130
Cursor class, 461
CursorMayhem.java, 462
cursors, 461
CustomerReporter.java, 512
custom packages
access control, 175
classes, adding, 175
folder structure, 174

naming, 173-174
cyclic gradients, 378

D

data, storing, 427
databases
accessing, 505
Apache Derby, 503
connecting to, 505-510
troubleshooting, 514
viewing connection information, 510
data source connections
closing, 512
opening, 508-510
drivers, 504-505, 508
Java DB, 503, 523
JDBC. *See* JDBC
queries, 504, 508-511
records
navigating, 511, 521-522
reading, 508-513
writing, 514-521
tables
creating, 517
viewing, 507-508
for XML-RPC servers, 564
DataInputStream class, 422
DataInputStream() constructor, 433
data input streams, 433
DataOutputStream class, 422
DataOutputStream() constructors, 433
data output streams, 433
data source connections

closing, 512
opening, 508-510
data streams, 433-436
data structures, 226
ArrayList class, 227, 233-235
arrays. *See* arrays
BitSet class, 227-232
Boolean arrays versus, 251
Enumeration interface, 227
generics, 246-250
HashMap class, 227, 241-246
Iterator interface, 227-229, 235-238
looping through, 235-238
Map interface, 240-241
Stack class, 227, 238-239
Vector class, 235
data types, 42-43
Apache XML-RPC support, 565
array elements, 98
boolean values, 42
casting, 83-84
characters, 42
char versus int data types, 445
enumerations, 249-250
floating-point numbers, 42
integers, 42
objects versus, 91
primitive, 42-43
to remote methods, 557
void, 43
XML-RPC support, 550
dateTime.iso8601 data type (XML-RPC), 550
DayCounter.java, 108

deallocating memory, 71
debugger (jdb), 642, 652-653
  advanced commands, 655-656
  applet debugging, 655
  application debugging, 653-655
debuggers (XML-RPC), 554
debugging, 652. *See also* troubleshooting
  advanced commands, 655-656
  applets, 655
  applications, 653-655
  breakpoints, 652
    deleting, 654
    setting, 653-655
  single-step execution, 652
declarations
  import, 171-172
  package, 175
declaring
  array variables, 96-97
  arrays of arrays, 102
  constants, 44
  interfaces, 176-179
  variables, 39-40
decode() method, 488
decrementing variables, 55-56
decrement operator (--), 55-56
default access, 159
  packages, 175
  protected access versus, 162
default package, 63
defining
  attributes, 17-18
  classes, 126

hierarchies, 169
instance variables, 21
methods, 21, 128-130
class methods, 134
this keyword, 130-131
variable scope, 131-132
subclasses, 26
values, shared, 43
variables
class variables, 74
instance variables, 126-127
delete() method, 442
deleting
breakpoints, 654
files, 442
deploying applications (Java Web Start), 392-395
configuring web servers for, 405
creating JNLP files, 396-404
description tag, 406
icon tag, 406-407
security, 405-406
@deprecated tag (javadoc), 650
deprecated methods, 642
description tag, 406
designing. *See also* creating
GUI in Android Studio, 581-584
server applications, 480-482
XML dialects, 528-529
destinations (casting), 83
development tools, selecting, 12-13, 616
device coordinate space, 378
dialects (XML), designing, 528-529
dialog boxes, 286

confirm dialog boxes, 287-288
example, 291-293
input dialog boxes, 288-289
message dialog boxes, 289
option dialog boxes, 290-291
diamond operator, 247
DiceRoller.java, 410
DiceWorker.java, 408
Dictionary class, 227
differentiating buttons in mouse events, 362
digital signatures, 404
Dimension class, 258
Dimension() constructor, 258
Dimension object, 264, 296
dimens.xml, 579
disabled components, 264
displaying frames, 258
dithering, 375
division operator, 52
DmozHandlerImpl.java, 563
DmozHandler.java, 562
DmozServer.java, 561
dockable toolbars, 298
!DOCTYPE declaration, 529
documentation
Java Class Library, 279
viewing, 47
documentation tool (javadoc), 646-650
Document object, creating, 533
Document Object Model (DOM), 530
documents
HTML, viewing, 643
XML

creating, 532-535
formatting, 540-542
modifying, 536-540
Document Type Definition (DTD), 529
doInBackground() method, 408
dollar sign ($) in variable names, 41
do loops, 118-119
DomainEditor.java, 538
DomainWriter.java, 541
DOM (Document Object Model), 530
dot notation, 72-73
calling class methods, 80
calling methods, 75
evaluating, 73
double ampersand (&&), AND operators, 57
double data type, 42
casting, 83
XML-RPC, 550
double pipe character (||), OR operators, 58
down command (jdb), 655
downloading
Apache XML-RPC, 554
JDK, 638
drawing
lines, 377
Line2D.Float class, 381
rendering attributes, 378-381
maps, 385-387
objects, 384
polygons, 377, 383-384
arcs, 382-383
ellipses, 382
rectangles, 381

rendering attributes, 378-381
strokes, 380-381
text, 370-372
antialiasing, 372
finding font information, 372-375
drawing colors, setting, 376-377
drawing surfaces, creating, 368-369
draw() method, 384
drawString() method, 370
DriverManager class, 509
drivers
for databases, 504-505, 508
USB, installing, 591
drop-down lists, 255, 274-276
DTD (Document Type Definition), 529
dynamic garbage collection, 72

E

Eclipse website, 13
editing. *See also* changing; modifying
system properties, 656-658
XML files, 576-577
editors (text). *See* text editors
Element object, creating, 533
elements (arrays)
accessing, 98-99, 233
changing, 99-102
data types, 98
grids, 102
elements (stack)
adding, 239
popping off, 239
searching, 239

elements (XML). *See* tags (XML)
Ellipse2D.Float class, 382
ellipses, drawing, 382
else keyword, 104
email address of author, 608
EML (Extended Machine Language), 546
empty() method, 239
empty statements in loops, 114
emulators. *See* Android Studio
enabling Intel Virtualization Technology in BIOS settings, 614
encapsulation, 159-161
enclosure tag, 528
encode() method, 488
endcap styles (drawing strokes), 380
ending. *See* stopping
end-of-line characters, 438, 441
Enumeration interface, 227
enumerations, 249-250
enum keyword, 249
environment variables, 656
EOFException (end-of-file exception), 195
  data streams, 434
  I/O streams, 422
equal sign (=) assignment operator, 40, 43, 54-55
equals() method, 88, 242
EqualsTester.java, 88
equal symbol (==) comparison operator, 57, 88
Error class, 194
error-handling. *See also* errors
  catching exceptions, 196
    finally clause, 199-202
    try and catch blocks, 196-199
  consistency checking, 195-196

passing exceptions, 204-205
throwing exceptions, 202, 207
checked, 204
inheritance, 206
nested handlers, 208-209
throws clause, 203
unchecked, 204
traditional method, 192-193
errors, 218. *See also* debugging; error-handling; exceptions; troubleshooting
Class not found, 633-635
compiler errors, 210
about generics, 251
for arrays, 99
runtime errors versus, 247
Error class, 194
Exception class, 194
fatal, troubleshooting, 196
NoClassDef, 633-635
runtime errors, compiler errors versus, 247
escape codes (Unicode character set), 49-50
evaluating
dot notation, 73
XOM, 542-545
event-handling, 339. *See also* event listeners
action events, 345-346
components, associating with event listeners, 341-342
focus events, 346-349
item events, 349-351
keyboard events, 351-352
methods, 342-345
mouse events, 352
mouse movement events, 352-357
window events, 357

event listeners, 330, 340
ActionListener, 340, 345
adapter classes and, 357-359
AdjustmentListener, 340
associating components with, 341-342
FocusListener, 340, 346
importing, 341
inner classes and, 359-362
ItemListener, 340
KeyListener, 340, 351
MouseListener, 340, 352
MouseMotionListener, 340, 352
property change listeners, 409
WindowListener, 340, 357, 455
events
action events, 340, 345-346
adjustment events, 340
focus events, 346-349
item events, 340, 349-351
keyboard events, 340, 351-352
keyboard focus events, 340
mouse events, 340, 352, 362
mouse movement events, 340, 352-357
window events, 340, 357
example code. *See* listings
Exception class, 194-195, 198
exception classes, constructors, 208
@exception tag (javadoc), 650
exceptions, 191-195. *See also* debugging; error-handling; errors; troubleshooting
ArrayIndexOutofBounds, 194
catching, 194-196
finally clause, 199-202

try and catch blocks, 196-199
checked, 195
compiler errors, 210
consistency checking, 195-196
creating, 207
EOFException, 195, 422, 434
Error class, 194
Exception class, 194-195
file operations, 442
inheritance, 207
InterruptedIOException, 475
IOException, 195, 489, 536
I/O streams, 421-422
limitations, 210
MalformedURLException, 195, 471
non-runtime, 219
NullPointerException, 194
ParseException, 536
passing, 204-205
runtime, 194, 219
RuntimeException, 195
SQLException, 509-511
Throwable class, 194-195
throwing, 194, 202, 207
checked exceptions, 204
inheritance issues, 206
nested handlers, 208-209
throws clause, 203
unchecked exceptions, 204
exclamation point (!), NOT operator, 58
exclusive radio buttons, 273
execute() method, 557
executeQuery() method, 510-511

exit command (jdb), 655
exiting loops, 119
expanding NetBeans panes, 604
explicit casts, 84
exponents in floating-point literals, 48
expressions, 51-52
  definition of, 38
  dot notation, 73
  grouping, 59
  operators. *See* operators
  readability, improving, 60
  return values, 38, 52
extending interfaces, 180-181
extends keyword, 126, 180
Extensible Markup Language. *See* XML
extensions
  .class, 639
  .java, 641

F

false value (Boolean), 49
fatal errors, troubleshooting, 196
feed2.rss, 540
FeedBar.java, 299
FeedBar2.java, 305
FeedInfo.java, 291
FileChannel objects, 488
File class, 441
file extensions. *See* extensions
FileInputStream class, 422
FileInputStream() constructor, 422
file input streams, 422-425
FileNotFoundException, 421

File objects, creating, 441
FileOutputStream class, 422
FileOutputStream() constructor, 425
file output streams, 425-427
FileReader class, 437
FileReader() constructor, 437
files
  archiving, 650-652
  compiling, 641-642
  deleting, 442
  JAR files, signing, 658-659
  JNLP files
    associating MIME types, 405
    creating, 396-404
    description tag, 406
    icon tag, 406-407
    security, 405-406
  multiple, compiling, 641
  Path object, 441-444
  relative paths, 444
  renaming, 442
  text files
    reading, 437-440
    writing, 440-441
  XML files, editing, 576-577
Files class, 442
FileSystems class, 441
FileWriter class, 440, 445
FileWriter() constructor, 440
fill() method, 384
fill patterns (Java2D), 378-380
filtering streams, 421, 427
FilterInputStream class, 427

FilterOutputStream class, 427
filters (streams), 421
final abstract methods, 187
final classes, 167-169
final keyword, 44
final methods, 167-168
final modifier, 158, 167
final variable, 167
finally statement, 199-202
finding font information, 372-375
Finger.java, 477
Finger protocol, 476
FingerServer.java, 495
finishConnect() method, 495
first() method, 521
flags, 227-232
flip() method, 485
float data type, 42, 83
floating-point numbers, 42
  exponents, 48
  hexadecimal numbers versus, 376
  Java2D, 388
  as literals, 48
floor() method, 280
FlowLayout class, 314-315
FlowLayout() constructor, 315
flow layout manager, 315-317
flush() method, 428
FocusAdapter class, 358, 457
focus events, 346-349
focusGained() method, 346
FocusListener event listener, 340, 346
focusLost() method, 346

folders
MS-DOS
creating, 622-623
opening, 621-622
structure (packages), 174
FontMetrics class, 373
Font objects, creating, 370-372
fonts
antialiasing, 372
built-in, 371
on buttons, changing, 281
finding information, 372-375
Font objects, creating, 370-372
styles, selecting, 371
for loops, 113-116
format commands (JDK), 638-639
FormatChooser.java, 349
FormatFrame.java, 273
FormatFrame2.java, 275
formatting
strings, 77-78
XML documents, 540-542
forName() method
character sets, 487
database drivers, 508
forward slash (/)
comment notation, 46
division operator, 52
forward slash equal (/=) assignment operator, 54
frames, 255-257
absolute component placement, 334
closing, 259-260
creating, 368

developing framework, 260-261
displaying, 258
locations, 258
resizing, 310
sizing, 257
visible, 258
frameworks (GUI), creating, 260-261
functional interfaces, 460
functional programming, 449, 465
functions of tools, modifying, 638

G

GeneralPath objects, creating, 384
generics, 246-251
getActionCommand() method, 345
getAppletInfo() method, 644-645, 655
getChannel() method, 488
getChar() method, 486
getChildElements() method, 537
getChild() method, 537
getClass() method, 90
getClickCount() method, 352
getColor() method, 377
getConnection() method, 509
getContentType() method, 474
getDate() method, 511
getDefault() method, 441
getDouble() method
byte buffers, 486
database records, 511
getErrorCode() method, 511
getFirstChildElement() method, 537
getFloat() method

byte buffers, 486
database records, 511
getFontMetrics() method, 373
getHeaderFieldKey() method, 474
getHeaderField() method, 474
getHeight() method, 373
getIcon() method, 267
getId() method, 586
getInsets() method, 334
getInt() method
byte buffers, 486
database records, 511
getItemAt() method, 275
getItemCount() method, 275
getItem() method, 349
getKeyChar() method, 351
getLong() method
byte buffers, 486
database records, 511
getMessage() method, 197
get() method
array lists, 233
buffers, 484
elements, 537
Map interface, 241
getParameterInfo() method, 644-645
getPath() method, 441
getPoint() method, 352
getProperties() method, 657
getProperty() method, 657
get requests, 551
getResponseCode() method, 474
getResponseMessage() method, 474

getRootElement() method, 536
getSelectedIndex() method, 275
getSelectedItem() method, 275
getSelectedText() method, 268
getSelectedValuesList() method, 277
getShort() method, 486
getSize() method, 264
getSource() method, 342, 345
getSQLState() method, 511
getStateChange() method, 349
getString() method, 511
getText() method, 267-268
getX() method, 352
getXmlRpcServer() method, 559
getY() method, 352
GiftShop.java, 185
GNU Lesser General Public License (LGPL), 531
Google, history of Android, 570
Gosling, James, 9-10, 571, 598
gradient fills, 378
graphical user interface. *See* GUI
graphics. *See also* image icons
  2D graphics. *See* Java2D
  in Android apps, 579
  Java2D graphics, casting objects, 85
  organizing in NetBeans, 266
Graphics2D class, 368
  coordinate system, 369-370
  creating drawing surface, 368-369
  drawing objects, 384
Graphics2D objects, casting, 85
Graphics class, 368
Graphics objects, casting, 85

greater than or equal to symbol (>=) comparison operator, 57
greater than symbol (>) comparison operator, 57
GridLayout class, 320
GridLayout() constructor, 320
grid layout manager, 320-321
grids, array elements, 102
grouping
  arguments, 137
  classes, 32
  expressions, 59
  interfaces, 32
  methods, 79
  packages in NetBeans, 183
guid tag, 527
GUI (graphical user interface)
  designing in Android Studio, 581-584
  Swing. *See* Swing

H

HalfDollars.java, 100
HalfLooper.java, 115
handling
  arguments in applications, 138-139
  errors. *See* error-handling
Hardware Accelerated Execution Manager. *See* HAXM
hardware requirements for HAXM, 612
Harold, Elliotte Rusty, 531, 545
hashCode() method, 242
HashMap class, 227, 241-246
hash maps, creating, 241
hasNext() method, 228
HAXM (Hardware Accelerated Execution Manager), 610
  checking BIOS settings, 614

installing, 611-613
requirements, 612
HelloUser.java, 630
helper classes, 136, 450
hexadecimal numbers, 49
floating-point literals versus, 376
HexReader.java, 200
hiding components, 264
hierarchies, 26-27
creating, 27-29
defining, 169
interface, 181
methods in, 30
history
of Android, 570-571
of Java, 10-11
HolidaySked.java, 231
homepage tag, 401
href attribute, 402
HTML documents, viewing, 643
HTTP, XML-RPC requests
responding to, 553-554
sending, 551-552
HttpUrlConnection class, 474
hyphen (-). *See* minus sign (-)

I

icon tag, 402, 406-407
IconFrame.java, 265
icons
for Android apps, 579-581
image icons, 255, 265-267
Java Web Start applications, 399

IDEA website, 13
identifying classes, 170
IDEs (integrated development environments)
  Android Studio. *See* Android Studio
  NetBeans. *See* NetBeans
  selecting, 12-13
if statements, 104-106
ignore command (jdb), 656
ImageButton widgets (Android), 582-584
image icons, 255, 265-267
ImageIcon objects, 265
implementing interfaces, 177-178
implements keyword, 177, 212
import declaration, 171-172
import statement, 32-34, 175, 256
importing
  Apache XML-RPC, 555
  classes, 171-173
  event listeners, 341
  packages, 171
improving
  application performance, 407-412
  readability
    of expressions, 60
    of programs, 46
increment operator (++), 55-56
incrementing variables, 55-56
increments in loops, 113
indexOf() method, 77, 235
index values of loops, 114
InetAddress class, 493
InetSocketAddress class, 493
Info command (appletviewer), 644

Info.java application, 292
information tag, 402
inheritance, 25-31
  access control, 163
  class hierarchies, creating, 27-29
  exceptions, creating, 207
  multiple, 31, 176
  single, 31, 176
  throwing exceptions, 206
initializing
  loops, 113
  objects, 71
inner classes, 359-362, 388, 450-453
  anonymous inner classes, 454-459, 466
  creating, 450
  scope, 450
input dialog boxes, 288-289
InputDialog class, 286-289
input/output. *See* I/O streams
InputStream class, 422
InputStreamReader class, 437
input streams, 371, 420. *See also* streams
  buffered input streams, 428
  console input streams, 431-432
  creating, 420
  data input streams, 433
  file input streams, 422-425
insertChild() method, 541
insert() method, 269
insets, 333-334
Insets class, 334
installing
  Android Studio, 571-572

Apache XML-RPC, 554-556
HAXM, 611-613
JDK (Java Development Kit), 616-619
JRE (Java Runtime Environment), 392
NetBeans, 598
InstanceCounter.java, 165
instance methods, 135. *See also* methods
instanceof operator, 59, 90, 343
instances, 15-16. *See also* objects
instance variables, 17, 39, 72
class variables versus, 33, 74
defining, 21, 126-127
initial values, 40
length, 99
nesting with method calls, 79
values
accessing, 72-73
modifying, 73-74
instantiation, 15
int data type, 42
casting, 83
char data type versus, 445
XML-RPC, 550
Integer class, 86
integer literals, 47-49
integers, data types, 42
integrated development environments. *See* IDEs
IntelliJ IDEA, 574
Intel Virtualization Technology, enabling in BIOS settings, 614
Intent class, 587
Intent() constructor, 587
intents, creating, 587
interface hierarchy, 181

interfaces, 31-32, 176
access control, 179
ActionListener, 330, 460
adapter classes, 357-359
classes versus, 176
command-line, 619-621
Comparable, 32
creating, 178-179
declaring, 176-179
Enumeration, 227
event listeners. *See* event listeners
extending, 180-181
functional interfaces, 460
grouping, 32
implementing, 177
Iterator, 227-229, 235-238
Map, 240-241
methods, 179-180
multiple interfaces, implementing, 177-178
objects, casting, 85
Paint, 378
Runnable, 191, 212
ScrollPaneConstants, 271, 297
SocketImplFactory, 480
Statement, 510
SwingConstants, 267, 294, 309
as variable type, 178
variables, 179
WindowListener, 455
interfaces (GUI)
creating, 257-259
event listeners. *See* event listeners
interface libraries, 310

wizard interface, 327
Internet Assigned Numbers Authority, 480
Internet connections. *See* networking
interpreter (java), 639-641
InterruptedIOException errors, 475
in variable (input stream), 431
invoking. *See* calling
I/O (input/output) streams, 419-420
buffered, 427-431
buffers, 484-486
byte streams, 420-422
channels, 488-491
character sets, 487-488
character streams, 420, 437-441
console input, 431-432
creating, 420
data streams, 433-436
exception handling, 421-422
file input streams, 422-425
file output streams, 425-427
filtering, 421, 427
nonblocking I/O network connections, 492-499
Path objects, 441-444
reading, 420
writing to, 421
IOException, 195, 489, 536
file operations, 442
I/O streams, 422
isAcceptable() method, 494
isCancelled() method, 413
isConnectible() method, 494
isConnectionPending() method, 495
isDone() method, 413

isEditable() method, 268
isEmpty() method, 241
ISO-LATIN-1 character set, 487
isReadable() method, 494
isWritable() method, 494
item events, 340, 349-351
Item.java, 181
ItemListener event listener, 340
ItemProp.java, 658
itemStateChanged() method, 349
Iterator interface, 227-229, 235-238
iterator() method, 235-238

J

j2se tag, 403
JAR files
  creating as JNLP files, 396-404
  signing, 658-659
jarsigner, 658-659
jar tag, 403
jar utility, 650-652
Java
  applications, 136
    compiling in Windows, 631-632
    creating, 135-136
    handling arguments, 138-139
    passing arguments to, 137
    running, 602-603, 631-632, 639
    sample program, 629-631
    speed, 642
  case-sensitivity, 41
  C++ versus, 11
  development tools, selecting, 12-13, 616

documentation, 47
explained, 11
fonts, built-in, 371
history of, 10-11
Java2D, 367
arcs, drawing, 382-383
casting objects, 85
color. *See* color
ellipses, drawing, 382
Graphics2D class, 368
coordinate system, 369-370
creating drawing surface, 368-369
lines
drawing, 377, 381
rendering attributes, 378-381
maps, drawing, 385-387
polygons
drawing, 377, 383-384
rendering attributes, 378-381
rectangles, drawing, 381
text
antialiasing, 372
drawing, 370-372
finding font information, 372-375
user versus device coordinate spaces, 378
java21days website, 23
Java API for XML Processing, 530
java.awt.color package, 375
java.awt.event package, 256, 340
ActionListener interface, 330, 460
adapter classes, 358, 456-457
event listeners, 455
java.awt.geom package, 381

java.awt package, 256
  BorderLayout class, 322
  CardLayout class, 326
  Color class, 32, 375
  Cursor class, 461
  FlowLayout class, 315
  Font class, 370
  FontMetrics class, 373
JavaBeans, 409
java.beans package, 409
javac compiler, 631, 641-642
Java Class Library, 16, 159, 225, 278-281
Java DB, 503
  Access and MySQL versus, 523
  connecting to, 505-510
JavaDB library, adding to projects, 512
Java Development Kit. *See* JDK
Javadoc comments, 47
javadoc documentation tool, 646-650
.java extensions, 641
.java files, associating with text editor, 630
/java folder (Android), 574
JavaFX, 310
java interpreter, 639-641
java.io package, 279, 419. *See also* streams
  File class, 441
  IOException class, 195
  PrintStream class, 79
java.io.tmpdir system property, 657
java.lang package, 32
  exception classes, 195
  Math class, 79
  primitive type classes, 86

Runnable interface, 212
System class, 79, 431, 657
Thread class, 211
Java Look and Feel Graphics Repository, 267
java.net package, 469-470. *See also* networking
InetAddress class, 493
InetSocketAddress class, 493
URL class, 471
java.nio.channels package, 484, 488
java.nio.charset package, 484, 487
java.nio.file package, 419
Files class, 442
Path objects, 441
java.nio package, 469, 484. *See also* I/O (input/output) streams
buffers, 484-486
channels, 488-491
nonblocking I/O network connections, 492-499
Java Plug-in, 393, 643
Java Runtime Environment (JRE), installing, 392
Java SE Development Kit 8, 12
java.sql package, 505. *See also* JDBC (Java Database Connectivity)
DriverManager class, 509
java.time package, 279
java.util package, 159. *See also* data structures
StringTokenizer class, 69
TimeZone class, 657
java.vendor system property, 657
java -version command, 624
java.version system property, 657
Java Virtual Machine (JVM), 11, 392, 601, 623
Java Web Start, 392-395
configuring web servers for, 405
description tag, 406

icon tag, 406-407
JNLP files, creating, 396-404
security, 405-406
javax.swing package, 159, 256
BoxLayout class, 317
JButton class, 16
JComponent class, 264
JPanel class, 325, 368
SwingConstants interface, 309
SwingWorker class, 407
javax.xml.parsers package, 530
JButton class, 16, 261
JCheckBox class, 272
JCheckBox() constructor, 272
JComboBox class, 274-275
JComboBox() constructor, 275
JComponent class, 256, 264
JDBC (Java Database Connectivity), 504-505
data source connections
closing, 512
opening, 508-510
databases, accessing, 505
drivers, 505
jdb debugger, 652-653
advanced commands, 655-656
applet debugging, 655
application debugging, 653-655
JDK (Java Development Kit), 12, 616, 637-638
command line, 638-639
configuring, 619
command-line interface, 619-621
creating folders, 622-623
opening folders, 621-622

running programs, 623
setting CLASSPATH variable, 633-636
setting PATH variable, 624-628
downloading, 638
installing, 616-619
system properties, 656-658
utilities
appletviewer browser, 642-646
jar, 650-652
jarsigner, 658-659
javac compiler, 641-642
javadoc documentation tool, 646-650
java interpreter, 639-641
jdb debugger, 652-656
keytool, 658-659
version number, 624
JDOM, 531
jEdit, 629
JFrame class, 257
JFrame() constructor, 257
JLabel class, 267
JLabel() methods, 267
JList class, 276
JList() constructor, 276
JMenuBar class, 303
JMenuBar() constructor, 305
JMenu class, 303
JMenuItem class, 303
JMenuItem() constructor, 303
JNLP files
associating MIME types, 405
creating, 396-404
description tag, 406

icon tag, 406-407
security, 405-406
jnlp tag, 402
JOptionPane class, 286
JPanel class, 262, 325, 368
JPasswordField class, 268
JProgressBar class, 300
JProgressBar() constructor, 301
JPython language, 640
JRadioButton class, 272
JRE (Java Runtime Environment), installing, 392
JRuby language, 640
JScrollBar class, 297
JScrollPane class, 271, 296
JScrollPane() constructor, 271, 296
JSlider class, 294
JSlider() constructor, 294
JTabbedPane class, 307
JTabbedPane() constructor, 307
JTextArea() constructors, 269
JTextComponent class, 268
JTextField class, 268
JTextField() constructor, 268
JToggleButton class, 272
JToolBar class, 297
JToolBar() constructor, 298
JudoScript language, 640
juncture styles (drawing strokes), 380
JUnit, 219
JVM (Java Virtual Machine), 11, 392, 601, 623

K

Key class, 388
KeyAdapter class, 358, 457
keyboard events, 340, 351-352
keyboard focus events, 340
KeyChecker.java, 358
KeyChecker2.java, 360
KeyListener event listener, 340, 351
key-mapped data structures
  Dictionary class, 227
  HashMap class, 241-246
  Map interface, 240-241
keyPressed() method, 351
keyReleased() method, 351
keystores, 658
keytool, 658-659
keyTyped() method, 351
keywords. *See also* commands; statements
  abstract, 169
  break, 107, 119-120
  case, 107
  class, 126, 450
  continue, 119-120
  else, 104
  enum, 249
  extends, 126, 180
  final, 44, 167
  implements, 177, 212
  modifiers. *See* modifiers
  new, 454
  null, 97
  private, 159-161

protected, 162
public, 161
return, 129
static, 24, 74, 126-127, 134, 164
super, 150
this, 130-131, 145, 327
throws, 203-205

L

labeled loops, 120
labels, 255, 267
aligning, 267
creating, 267
menus, 304
progress bars, 301
sliders, 294-295
lambda expressions, origin of term, 466. *See also* closures
languages
JPython, 640
JRuby, 640
JudoScript, 640
NetRexx, 640
SQL (Structured Query Language), 504-505
lastElement() method, 233
last() method, 522
layout managers, 314-315
alternatives to, 334
border layout, 322-324
box layout, 317-319
card layout, 325-333
combining, 324-325
creating, 314
flow layout, 315-317

grid layout, 320-321
insets, 333-334
length instance variable, 99
length() method, 76, 91
less than or equal to symbol (<=) comparison operator, 57
less than symbol (<) comparison operator, 57
lexical scope, 121
LGPL (GNU Lesser General Public License), 531
libraries
Apache Project, 279
for interfaces (GUI), 310
Java Class Library. *See* Java Class Library
XOM. *See* XOM
licensing
Open Directory License, 559
for XOM, 531
Line2D.Float class, 381
line numbers in text editors, 629
lines, drawing, 377
Line2D.Float class, 381
rendering attributes, 378-381
lineTo() method, 384
linked lists, 91
linking
applets, URL objects, 471
node objects, 91
list command (jdb), 654
listeners. *See* event listeners
listings
AllCapsDemo.java, 442
Alphabet.java, 316
AppInfo.html, 645
AppInfo.java, 644

AppInfo2.java, 647
ArrayCopier.java, 117
Authenticator.java, 269
Authenticator2.java, 272
Averager.java, 138
Border.java, 323
Box.java, 141
Box2.java, 146
BufferConverter.java, 490
BufferDemo.java, 429
Bunch.java, 320
ButtonFrame.java, 262
Buttons.java, 263
ByteReader.java, 424
ByteWriter.java, 426
Calculator.java, 347
ClosureMayhem.java, 464
CodeKeeper.java, 236
CodeKeeper2.java, 248
ComicBooks.java, 243
ComicBox.java, 452
ConsoleInput.java, 432
CursorMayhem.java, 462
CustomerReporter.java, 512
DayCounter.java, 108
DiceRoller.java, 410
DiceWorker.java, 408
DmozHandlerImpl.java, 563
DmozHandler.java, 562
DmozServer.java, 561
DomainEditor.java, 538
DomainWriter.java, 541
EqualsTester.java, 88

feed2.rss, 540
FeedBar.java, 299
FeedBar2.java, 305
FeedInfo.java, 291
feed.rss, 532
Finger.java, 477
FingerServer.java, 495
FormatChooser.java, 349
FormatFrame.java, 273
FormatFrame2.java, 275
GiftShop.java, 185
HalfDollars.java, 100
HalfLooper.java, 115
HelloUser.java, 630
HexReader.java, 200
HolidaySked.java, 231
IconFrame.java, 265
Info.java application, 292
InstanceCounter.java, 165
Item.java, 181
ItemProp.java, 658
KeyChecker.java, 358
KeyChecker2.java, 360
Map.java, 385
MarsApplication.java, 23
MarsRobot.java, 20
MousePrank.java, 353
NamedPoint class, 151
PageData.java, 396
PageData.jnlp, 400
Passer.java, 133
PointSetter.java, 73
PrimeFinder.java, 213

PrimeReader.java, 435
PrimeThreads.java, 215
PrimeWriter.java, 434
Printer.java, 148
ProgressMonitor.java, 302
ProgressMonitor2.java, 458
QuoteData.java, 518
RangeLister.java, 129
RefTester.java, 80
RssFilter.java, 543
RssStarter.java, 534
SantaActivity.java
    full text, 588
    starting text, 585
SimpleFrame.java, 260
SiteClient.java, 558
Slider.java, 295
SourceReader.java, 439
Spartacus.java, 601
Stacker.java, 318
Storefront.java, 184
StringChecker.java, 76
Subscriptions.java, 277
SurveyFrame.java, 333
SurveyWizard.java, 331
TabPanels.java, 308
TextFrame.java, 373
TimeServer.java, 481
TitleBar.java, 343
TokenTester.java, 69
Variables.java, 45
Weather.java, 53
WebReader.java, 471

workbench.rss, 526
XML-RPC request, 552
XML-RPC response, 553
lists, 276-278
literals, 47
Boolean, 49
character, 49-50
integer, 47-49
string, 50-51
load factor (hash maps), 241
loading
classes, 508
database drivers, 508
locals command (jdb), 654
local scope, 131
local variables, 39
locations, frames, 258
logical fonts, 371
logical operators, 57-58
long data type, 42, 83
look and feel, 256
looping through data structures, 235-238
loops
breaking, 119
do, 118-119
for, 113-116
increments, 113
index values, 114
initialization, 113
labeling, 120
restarting, 119-120
run() method, stopping threads, 217-218
tests, 113

while, 116-118

M

main-class attribute, 403
main classes, designating, 136
main() method, 24, 135-136, 639
importance of, 603
as public, 161
MalformedURLException, 195, 471
managing
errors. *See* error-handling
exceptions. *See* error-handling
memory, 71-72
manifest files, configuring in Android Studio, 581
/manifests/AndroidManifest.xml, 574
Map interface, 240-241
Map.java, 385
maps, drawing, 385-387
MarsApplication.java, 23
MarsRobot.java, 20
Math class, 79, 280
math operators. *See* arithmetic operators
MD command (MS-DOS), 622
member variables. *See* instance variables
memory
allocating, 71
deallocating, 71
managing, 71-72
reclaiming, 72
memory command (jdb), 656
menu commands, appletviewer browser, 643-644
menus, 303-307
creating, 304

labels, 304
separators, adding, 304
message dialog boxes, 289
MessageDialog class, 286, 289
methods, 18
abstract methods, 169
accept(), 479
access control, 159
accessor methods, 164, 187
comparison of types, 163
default access, 159
inheritance, 163
private access, 159-161
protected access, 162
public access, 161
actionPerformed()
action events, 342, 345
buttons, 330
adapter classes, 357-359
add()
array lists, 233-234
border layouts, 323
card layouts, 326
check boxes/radio buttons, 273
containers, 262
menus, 304
addActionListener(), 330, 341, 345
addAttribute(), 533
addFocusListener(), 341
addHandler(), 560
addItem(), 275
addItemListener(), 341
addMouseListener(), 341

addMouseMotionListener(), 341
addSeparator(), 304
addTab()panes, 307
addTextListener(), 341
addWindowListener(), 341
afterLast(), 521
allocate(), 485
append(), 269
appendChild(), 533
beforeFirst(), 521
build(), 536
calling, 18, 75-79
chaining, 655
channel(), 495
charAt(), 77, 91
charWidth(), 373
in class hierarchy, 30
class methods, 19, 79-80, 134-135
  accessing, 165
  calling, 80
  defining, 134
clear()
  array lists, 235
  hash maps, 242
close()
  buffered character streams, 441
  character streams, 440
  client-side sockets, 476
  data source connections, 512
  data streams, 434
  file output streams, 425
  streams, 420-421
closePath(), 384

configureBlocking(), 493
connect(), 493
constructors, 69, 144-145
  calling, 144, 151
  calling from another constructor, 145-146
  definition of, 71
  naming, 144
  overloading, 146-147
  overriding, 150-152
contains(), 235
containsKey(), 242
containsValue(), 242
createFont(), 371
createStatement(), 510, 522
decode(), 488
defining, 21, 128-130
  this keyword, 130-131
  variable scope, 131-132
delete(), 442
deprecated methods, 642
doInBackground(), 408
draw(), 384
drawString(), 370
empty(), 239
encode(), 488
equals(), 88, 242
event-handling methods, 342-345
execute(), 557
executeQuery(), 510-511
fill(), 384
final abstract methods, 187
final methods, 167-168
finishConnect(), 495

first(), 521
flip(), 485
floor(), 280
flush(), 428
focusGained(), 346
focusLost(), 346
forName()
  character sets, 487
  database drivers, 508
get()
  array lists, 233
  buffers, 484
  elements, 537
  Map interface, 241
getActionCommand(), 345
getAppletInfo(), 644-645, 655
getChannel(), 488
getChar(), 486
getChild(), 537
getChildElements(), 537
getClass(), 90
getClickCount(), 352
getColor(), 377
getConnection(), 509
getContentType(), 474
getDate(), 511
getDefault(), 441
getDouble()
  byte buffers, 486
  database records, 511
getErrorCode(), 511
getFirstChildElement(), 537
getFloat()

byte buffers, 486
database records, 511
getFontMetrics(), 373
getHeaderField(), 474
getHeaderFieldKey(), 474
getHeight(), 373
getIcon(), 267
getId(), 586
getInsets(), 334
getInt()
byte buffers, 486
database records, 511
getItem(), 349
getItemAt(), 275
getItemCount(), 275
getKeyChar(), 351
getLong()
byte buffers, 486
database records, 511
getMessage(), 197
getParameterInfo(), 644-645
getPath(), 441
getPoint(), 352
getProperties(), 657
getProperty(), 657
getResponseCode(), 474
getResponseMessage(), 474
getRootElement(), 536
getSelectedIndex(), 275
getSelectedItem(), 275
getSelectedText(), 268
getSelectedValuesList(), 277
getShort(), 486

getSize(), 264
getSource(), 342, 345
getSQLState(), 511
getStateChange(), 349
getString(), 511
getText(), 267-268
getX(), 352
getXmlRpcServer(), 559
getY(), 352
grouping, 79
hashCode(), 242
hasNext(), 228
indexOf(), 77, 235
insert(), 269
insertChild(), 541
instance methods, 135
in interfaces, 31, 179-180
isAcceptable(), 494
isCancelled(), 413
isConnectible(), 494
isConnectionPending(), 495
isDone(), 413
isEditable(), 268
isEmpty(), 241
isReadable(), 494
isWritable(), 494
itemStateChanged(), 349
iterator(), 235-238
JLabel(), 267
keyPressed(), 351
keyReleased(), 351
keyTyped(), 351
last(), 522

lastElement(), 233
length(), 76, 91
lineTo(), 384
main(), 24, 135-136, 639
  importance of, 603
  as public, 161
mouseClicked(), 352, 362
mouseDragged(), 353
mouseEntered(), 352
mouseExited(), 352
mouseMoved(), 353
mousePressed(), 352
mouseReleased(), 352
move(), 442
moveTo(), 384
newDecoder(), 488
newEncoder(), 488
newLine(), 441
next()
  Iterator interface, 228
  resultsets, 521
  socket channels, 494
onCreate(), 586
open(), 493
overloading, 128
  advantages, 139
  creating overloaded methods, 140-143
  definition of, 139
  troubleshooting, 140
overriding, 30-31, 147-149
  advantages, 149-150
  super keyword, 150
pack(), 258, 310

paintComponent(), 368, 377
parameter lists, 128
parseInt(), 86
passing arguments to, 132-134
peek(), 239
pop(), 239
position(), 485
prepareStatement(), 514
previous(), 522
print(), 46
println(), 46, 79
printStackTrace(), 199
private abstract methods, 187
private methods as final, 168
processClicks(), 586
protecting, 170
push(), 239
put()
  buffers, 485
  Map interface, 240
putChar(), 486
putDouble(), 486
putFloat(), 486
putInt(), 486
putLong(), 486
putShort(), 486
random(), 280
read()
  buffered character streams, 438
  buffered input streams, 428
  byte buffers, 489
  character streams, 437
  file input streams, 423

filters, 421
streams, 420
readBoolean(), 433
readByte(), 433
readDouble(), 433
readFloat(), 433
readInt(), 433
readLine(), 438
readLong(), 433
readShort(), 433
readUnsignedByte(), 433
readUnsignedShort(), 433
register(), 493
remove()
array lists, 234-235
Map interface, 241
socket channels, 495
removeChild(), 538
requestFocus(), 346
return types, 128-129
run(), 213, 217-218
search(), 239
select(), 494
selectedKeys(), 494
set(), 234
setActionCommand(), 346
setAsciiStream(), 515
setBackground(), 377
setBinaryStream(), 515
setBoolean(), 515
setBounds()
components, 335
frames, 258

setByte(), 515
setBytes(), 515
setCharacterStream(), 515
setColor(), 376
setConfig(), 556
setContentView(), 586
setCursor(), 461
setDate(), 515
setDefaultCloseOperation(), 259
setDouble(), 515
setEchoChar(), 268
setEditable()
  combo boxes, 275
  text fields, 268
setEnabled(), 264
setFloat(), 515
setFollowRedirects(), 474
setFont(), 281, 371
setHgap(), 320
setIcon(), 267
setIndentation(), 541
setInt(), 515
setJMenuBar(), 305
setLayout()
  card layouts, 326
  containers, 314
  panels, 325
setLineWrap(), 269
setListData(), 277
setLong(), 515
setLookAndFeel()
  dialog boxes, 293
  GUIs, 259

setMajorTickSpacing(), 294
setMaximum(), 301
setMinimum(), 301
setMinorTickSpacing(), 294
setNull(), 516
setPaint(), 378
setPaintLabels(), 295
setPaintTicks(), 295
setPreferredSize(), 296
setRenderingHint(), 372
setSelected(), 272
setSelectedIndex(), 275
setServerURL(), 556
setShort(), 515
setSize()
    components, 264
    frames, 257
setSoTimeOut(), 475
setString(), 515
setStringPainted(), 301
setStroke(), 380
setText(), 267-268
setValue(), 301
setVgap(), 320
setVisible()
    components, 264
    frames, 258
setVisibleRowCount(), 277
setWrapStyleWord(), 269
show(), 326
showConfirmDialog(), 287
showInputDialog(), 288
showMessageDialog(), 289

showOptionDialog(), 290
signatures, 128, 139
size()
  array lists, 235
  elements, 537
  file channels, 489
  Map interface, 241
start(), 212
startActivity(), 587
static, 164-167
stringWidth(), 373
substring(), 77, 201
super(), 151
  in superclasses, calling, 150
System.out.format(), 77-78
throwing exceptions, 202, 207
  checked, 204
  inheritance, 206
  nested handlers, 208-209
  throws clause, 203
  unchecked, 204
toFile(), 441
toPath(), 442
toUpperCase(), 77
toXML(), 534
trimToSize(), 235
valueOf(), 79
windowActivated(), 357
windowClosed(), 357
windowClosing(), 357
windowDeactivated(), 357
windowDeiconified(), 357
windowIconified, 357

windowOpened(), 357
wrap(), 484
write()
buffered character streams, 441
buffered output streams, 428
character streams, 440
char versus int data types, 445
file output streams, 425
filters, 421
streams, 421
XML documents, 541
writeBoolean(), 433
writeByte(), 433
writeDouble(), 433
writeFloat(), 433
writeInt(), 433
writeLong(), 433
writeShort(), 433
in XML-RPC, 552
zero-based, 77
methods command (jdb), 656
Microsoft Word, 629
MIME types, associating, 405
minus equal (-=) assignment operator, 54
minus sign (-)
decrement operator (--), 55-56
negative numbers, 48
subtraction operator, 52
modifiers, 158
abstract, 169
access control, 159
accessor methods, 164
comparison of types, 163

default access, 159
inheritance, 163
private access, 159-161
protected access, 162
public access, 161
final, 167
multiple, 158
private, 159-161
protected, 162
public, 161
return types versus, 158
static, 164
modifying. *See also* changing; editing
class variable values, 75
functions, 638
instance variable values, 73-74
operator precedence, 60
superclasses, 27
XML documents, 536-540
modulus operator, 52
MouseAdapter class, 358, 457
mouseClicked() method, 352, 362
mouseDragged() method, 353
mouseEntered() method, 352
MouseEvent objects, 352
mouse events, 340, 352, 362
mouseExited() method, 352
MouseListener, 340, 352
MouseMotionAdapter class, 358
MouseMotionListener, 340, 352
mouseMoved() method, 353
mouse movement events, 340, 352-357
MousePrank.java, 353

mousePressed() method, 352
mouseReleased() method, 352
move() method, 442
moveTo() method, 384
MS-DOS, 620
  CLASSPATH variable
    Windows 7-10, 633-635
    Windows 98/Me, 635-636
  commands, 621-622
  folders
    creating, 622-623
    opening, 621-622
  PATH variable
    Windows 7-10, 625-627
    Windows 98/Me, 627-628
  programs, running, 623
  prompt. *See* command line
multidimensional arrays, 102
multiline comments, 46
multiple bytes, writing, 425
multiple files, compiling, 641
multiple inheritance, 31, 176
multiple interfaces, implementing, 177-178
multiple modifiers, 158
multiplication operator, 52
multitasking. *See* threads
MySQL databases, Java DB versus, 523

N

NamedPoint class, 151
naming
  class variables, 127
  conflicts, reducing, 170-172

constants, 44
methods, constructors, 144
packages, 173-174
resources, 580
variables, 40-41
navigating
database records, 521-522
records, 511
negative numbers, as literals, 48
nesting
exception handlers, 208-209
if statements, 106
method calls, 78-79
XML tags, 527
NetBeans, 12, 597
adding
Apache XML-RPC to, 555
JavaDB library to projects, 512
XOM to, 532
connecting to databases, 505-508
database connection information, viewing, 510
database tables, creating, 517
grouping packages, 183
installing, 598
Java applications, running, 602-603
Java classes
compiling, 601
creating, 600-602
importing, 173
main classes, designating, 136
organizing graphics, 266
panes, expanding/shrinking, 604
projects, creating, 19, 598-600

resources for information, 605
Run File versus Run Project commands, 134
setting command-line arguments, 109
troubleshooting in, 603
updating, 598
NetRexx language, 640
networking, 470
nonblocking I/O connections, 492-499
security, 499
sockets, 475-479
client-side, 475-476
server-side, 479-483
streams, 470-475
web services, 549
XML-RPC. *See* XML-RPC
newDecoder() method, 488
newEncoder() method, 488
new keyword, 454
newLine() method, 441
new operator, 59, 68
creating objects with, 68-70
instantiating arrays, 97
next() method
Iterator interface, 228
resultsets, 521
socket channels, 494
NoClassDef error, 633-635
Node class, 533
node objects, linking, 91
nodes, adding children to parents, 533
nonblocking I/O network connections, 492-499
nonexclusive check boxes, 273
non-runtime exceptions, 219

NoSuchFileException, 442
NotePad, 628
not equal symbol (!=) comparison operator, 57, 88
NOT operator, 58
null keyword, 97
NullPointerException, 194
number literals, 47-49
numbers
  binary, 48
  converting strings to, 86
  floating-point, 42
  formatting display, 77-78
  hexadecimal, 49
  integers, 42
  octal, 48
nu.xom.canonical package, 542
nu.xom.converters package, 542
nu.xom package, 533
nu.xom.xinclude package, 542
nu.xom.xslt package, 542

O

Object class, 26
object-oriented programming (OOP), 11-14. *See also* classes; objects
objects, 14. *See also* instances
  ArrayList, 556
  arrays, creating, 97-98
  attributes, 17-18
    in class hierarchies, 29
    defining, 17
  behavior, 18-19
  ButtonGroup, 273
  ByteBuffer, 489

casting, 82-85, 565
classes and, 14-16
Color, creating, 376
comparing, 87-89
creating, 68
  arguments, 68
  with closures, 460-465
  with constructors, 71
  with new operator, 68-70
  StringTokenizer objects, 69-70
current, referring to, 130
determining class of, 89-90
Dimension, 264, 296
Document, 533
drawing, 384
Element, creating, 533
encapsulating, 161
File, creating, 441
FileChannel, 488
Font, creating, 370-372
GeneralPath, creating, 384
ImageIcon, 265
inheritance, 29-31
initializing, 71
memory, allocating/deallocating, 71
MouseEvent, 352
nodes, linking, 91
Path, 441-444
primitive types, 91
  converting, 86-87
Rectangle, frame boundaries, 258
references, 80-82
ResultSet, 511

reusing, 15-16
Selector, 493
Serializer, 540
Set, 494
String, creating, 21
URL, creating, 471
object variables. *See* instance variables
object wrappers, 86
obscuring password fields, 268
octal numbers, 48
offline-allowed tag, 401
onCreate() method, 586
online storefronts, creating, 181-187
OOP (object-oriented programming), 11-14. *See also* classes; objects
Open Directory License, 559
Open Directory Project, 559
opening
data source connections, 508-510
folders in MS-DOS, 621-622
socket connections, 475
sockets, 493
streams over Internet, 470-475
opening tags (XML), 401, 527
open() method, 493
operators, 52
arithmetic, 52-54
assignment, 40, 43, 54-55
bitwise, 59
comparison, 56-57, 88
concatenation, 82
decrement (--), 55-56
diamond operator, 247
increment (++), 55-56

instanceof, 59, 90, 343
list of, 61
logical, 57-58
new, 59, 68
creating objects with, 68-70
instantiating arrays, 97
postfix, 55-56
precedence, 58-60
prefix, 55-56
string concatenation, 60-61
ternary, 59, 112
option dialog boxes, 290-291
OptionDialog class, 286, 290-291
options (commands), 638-639
order of precedence, 58-60
organizing
classes, 25, 158, 170
creating hierarchies, 27-29
inheritance, 25-31
interfaces, 31-32
packages, 32, 45
graphics in NetBeans, 266
projects in Android Studio, 574-575
org.apache.xmlrpc package, 554
org.apache.xmlrpc.client package, 556
org.apache.xmlrpc.server package, 559
org.apache.xmlrpc.webserver package, 559
orientation
progress bars, 301
sliders, 294
toolbars, 297
OR operators, 58
os.name system property, 657

os.version system property, 657
OutputStream class, 422
output streams, 420. *See also* streams
  buffered output streams, 428
  creating, 420
  data output streams, 433
  file output streams, 425-427
OutputStreamWriter class, 440
overflow (variable assignment), 62
overloading
  constructors, 146-147
  methods, 128
    advantages, 139
    creating, 140-143
    definition of, 139
    troubleshooting, 140
overriding
  constructors, 150-152
  methods, 30-31, 147-150
  scrollbars, 297

P

package declaration, 175
package statement, 32, 63
packages, 32, 45, 169
  access control, 175
  advantages, 170
  android.content, 587
  android.support.v7.app, 586
  creating, 173-175, 640
  grouping in NetBeans, 183
  importing, 171
  java.awt, 256

BorderLayout class, 322
CardLayout class, 326
Color class, 32, 375
Cursor class, 461
FlowLayout class, 315
Font class, 370
FontMetrics class, 373
java.awt.color, 375
java.awt.event, 256, 340
ActionListener interface, 330, 460
adapter classes, 358, 456-457
event listeners, 455
java.awt.geom, 381
java.beans, 409
java.io, 279, 419
File class, 441
IOException class, 195
PrintStream class, 79
java.lang, 32
exception classes, 195
Math class, 79
primitive type classes, 86
Runnable interface, 212
System class, 79, 431, 657
Thread class, 211
java.net, 469-470
InetAddress class, 493
InetSocketAddress class, 493
URL class, 471
java.nio, 469, 484
buffers, 484-486
channels, 488-491
nonblocking I/O network connections, 492-499

java.nio.channels, 484, 488
java.nio.charset, 484, 487
java.nio.file, 419, 441-442
java.sql, 505, 509
java.time, 279
java.util, 159. *See also* data structures
StringTokenizer class, 69
TimeZone class, 657
javax.swing, 159, 256
BoxLayout class, 317
JButton class, 16
JComponent class, 264
JPanel, 368
JPanel class, 325
SwingConstants interface, 309
SwingWorker class, 407
javax.xml.parsers, 530
nu.xom, 533
nu.xom.canonical, 542
nu.xom.converters, 542
nu.xom.xinclude, 542
nu.xom.xslt, 542
org.apache.xmlrpc, 554
org.apache.xmlrpc.client, 556
org.apache.xmlrpc.server, 559
org.apache.xmlrpc.webserver, 559
referencing, 170
pack() method, 258, 310
PageData.java, 396
PageData.jnlp, 400
paintComponent() method, 368, 377
Paint interface, 378
panels, 262, 325

absolute component placement, 334
card layouts, 326
components, adding, 325
creating, 325, 368
insets, 333-334
scrolling panes, 255, 271-272, 296-297
tabbed panes, 307-310
panes (NetBeans), expanding/shrinking, 604
@param tag (javadoc), 650
parameter lists (methods), 128
parameters (XML-RPC), 553
parentheses ()
arguments, 68
grouping expressions, 59-60
parent nodes, adding child nodes to, 533
ParseException errors, 536
parseInt() method, 86
Passer.java, 133
passing
arguments
to applications, 137
to methods, 132-134
exceptions, 204-205
password fields, obscuring, 268
Path objects, 441-444
paths, relative, 444
PATH variable (MS-DOS)
Windows 7-10, 625-627
Windows 98/Me, 627-628
peek() method, 239
percent sign (%) modulus operator, 52
performance
improving, 407-412

Java programs, 642
period (.)
accessing methods and variables, 59
dot notation, 73
permalinks, 527
pipe character (|) OR operator, 58
platform neutrality, 11
plus equal (+=) assignment operator, 54
plus sign (+)
addition operator, 52
concatenation operator, 82
increment operator (++), 55-56
string concatenation, 60-61
pointers (C/C++), 82, 91. *See also* arrays; references
PointSetter.java, 73
polygons, drawing, 377, 383-384
arcs, 382-383
ellipses, 382
rectangles, 381
rendering attributes, 378-381
pop() method, 239
port numbers, selecting, 480
position() method, 485
postfix operators, 55-56
post requests, 551
precedence of operators, 58-60
prefix operators, 55-56
prepared statements, 514-516
PreparedStatement class, 514
prepareStatement() method, 514
preparing resources in Android Studio, 579-580
previous() method, 522
PrimeFinder.java, 213

PrimeReader.java, 435
PrimeThreads.java, 215
PrimeWriter.java, 434
primitive types, 42-43
  casting, 82-84
  objects, 91
    converting, 86-87
print command (jdb), 654
Printer.java, 148
println() method, 46, 79
print() method, 46
printStackTrace() method, 199
PrintStream class, 79
private abstract methods, 187
private access, 159-161
private methods, as final, 168
private modifier, 158-161
problem-solving. *See* troubleshooting
procedural programming, 13
procedures (XML-RPC), 553
processClicks() method, 586
processing XML
  with Java, 530
  with XOM, 530-532
    creating XML documents, 532-535
    evaluating XOM, 542-545
    formatting XML documents, 540-542
    modifying XML documents, 536-540
programming
  object-oriented, 11-14
  procedural, 13
programs
  classes versus, 125

Java. *See* Java applications
MS-DOS, running, 623
readability, improving, 46
running, 22-25
progress bars, 300-303
labels, 301
orientation, 301
updating, 301
ProgressMonitor.java, 302
ProgressMonitor2.java, 458
projects
adding JavaDB library to, 512
closing in Android Studio, 579
creating, 19
in Android Studio, 572-574, 579
in NetBeans, 598-600
organizing in Android Studio, 574-575
properties, system, 656-658
property change listeners, 409
PropertyHandlerMapping class, 560
protected access, 162
protected modifier, 158, 162
protecting classes/methods/variables, 170. *See also* access control
public access, 161, 175
public modifier, 158, 161
push() method, 239
putChar() method, 486
putDouble() method, 486
putFloat() method, 486
putInt() method, 486
putLong() method, 486
put() method
buffers, 485

Map interface, 240
putShort() method, 486

Q

queries, 504, 508-511
question mark (?) in SQL statements, 514
quitting. *See* stopping
quotation marks (“”) in arguments, 137
QuoteData.java, 518

R

radio buttons, 255, 272-274
event handling
action events, 345-346
item events, 349-351
exclusive, 273
random() method, 280
RangeLister.java, 129
RDF Site Summary, 528
readability, improving
expressions, 60
programs, 46
readBoolean() method, 433
readByte() method, 433
readDouble() method, 433
Reader class, 437
readFloat() method, 433
reading
buffered character streams, 438
buffered input streams, 428
C programs, 444
database records, 508-513
data input streams, 433
streams, 420

text files, 437-440
readInt() method, 433
readLine() method, 438
readLong() method, 433
read() method
buffered character streams, 438
buffered input streams, 428
byte buffers, 489
character streams, 437
file input streams, 423
filters, 421
streams, 420
readShort() method, 433
readUnsignedByte() method, 433
readUnsignedShort() method, 433
Really Simple Syndication. *See* RSS
reclaiming memory, 72
records
in databases
navigating, 521-522
reading, 508-513
writing, 514-521
navigating, 511
Rectangle2D.Float class, 382
Rectangle objects, frame boundaries, 258
rectangles, drawing, 381
reducing name conflicts, 170
references, 80-82
arrays, 99
to current objects, 130
passing arguments by, 132
to packages, 170
RefTester.java, 80

register() method, 493
relative paths, 444
Reload command (appletviewer), 643
remote method invocation (RMI), 550
remote procedure calls (RPC), 549-550. *See also* XML-RPC
removeChild() method, 538
remove() method
  array lists, 234-235
  Map interface, 241
  socket channels, 495
removing stack elements, 239
renaming files, 442
rendering attributes (Java2D), 378-381
RenderingHint.Key class, 388
requestFocus() method, 346
requests (XML-RPC)
  responding to, 553-554
  sending, 551-552
requirements for HAXM, 612
/res folder (Android), 574, 579
/res/layout folder (Android), 582
/res/mipmap folder (Android), 579
reserved words. *See* keywords; modifiers
resizing
  components, 264
  frames, 310
  NetBeans panes, 604
  scrolling panes, 296
resources
  for information
    author contact information, 608
    book website, 607
    NetBeans, 605

naming, 580
preparing in Android Studio, 579-580
resources tag, 402
responding to XML-RPC requests, 553-554
Restart command (appletviewer), 643
restarting loops, 119-120
ResultSet object, 511
resultsets, navigating, 521-522
return keyword, 129
@return tag (javadoc), 647
return types, 128
methods, void, 129
modifiers versus, 158
return values, 38, 52
reusing objects, 15-16
R.java class, 586
RMI (remote method invocation), 550
RPC (remote procedure calls), 549-550. *See also* XML-RPC
RSS (Really Simple Syndication), 525, 528
evaluating XOM, 542-545
versions 1.0 and 2.0, 546
well-formed XML, 528
XML and, 526
XML documents
creating, 532-535
formatting, 540-542
modifying, 536-540
RSS Advisory Board, 528
RssFilter.java, 543
RssStarter.java, 534
run command (jdb), 654
Run File command, Run Project command versus, 134
run() method, 213, 217-218

Runnable interface, 191, 212
running
  Android apps, 577-578, 589-591
  applications, 639
  bytecode, 639
  Java applications, 602-603
    in Windows, 631-632
  JVM, 623
  programs, 22-25
    in MS-DOS, 623
  self-signed JAR files, 659
  telnet, 482
  threads, 213
Run Project command, Run File command versus, 134
runtime errors, compiler errors versus, 247
RuntimeException class, 195
runtime exceptions, 194, 204, 219

S

Sams Teach Yourself Android Application Development in 24 Hours (Delessio, Darcey, Conder), 589
Sams Teach Yourself SQL in 24 Hours (Stephens, Jones, Plew), 510
SantaActivity.java
  full text, 588
  starting text, 585
saving source files, 630
SAX (Simple API for XML), 530
scope
  inner classes, 450
  lexical scope, 121
  variables, 103, 131-132
scrollbars, 296
  configuring, 271

overriding, 297
scrolling
panes, 255, 271-272, 296-297
tabbed panes, 307
ScrollPaneConstants interface, 271, 297
searching stacks, 239
search() method, 239
security
digital signatures, 404
Java DB, 506
Java Web Start applications, 394-395
networking, 499
SecurityException, 442
security tag, 405-406
@see tag (javadoc), 650
selectedKeys() method, 494
selecting
development tools, 12-13, 616
font styles, 371
port numbers, 480
substrings, 201
text editors, 628-629
SelectionKey class, 494
select() method, 494
Selector object, 493
self-signed JAR files, running, 659
semicolon (;), statement termination character, 38
sending XML-RPC requests, 551-552
separators (menus), adding, 304
@serial tag (javadoc), 647
Serializer class, 540
Serializer() constructor, 541
servers, XML-RPC, 559-564

server-side sockets, 479-480
  designing server applications, 480-482
  nonblocking servers, 493-499
  testing server applications, 482-483
ServerSocket class, 479
setActionCommand() method, 346
setAsciiStream() method, 515
setBackground() method, 377
setBinaryStream() method, 515
setBoolean() method, 515
setBounds() method
  components, 335
  frames, 258
setByte() method, 515
setBytes() method, 515
setCharacterStream() method, 515
setColor() method, 376
setConfig() method, 556
setContentView() method, 586
setCursor() method, 461
setDate() method, 515
setDefaultCloseOperation() method, 259
setDouble() method, 515
setEchoChar() method, 268
setEditable() method
  combo boxes, 275
  text fields, 268
setEnabled() method, 264
setFloat() method, 515
setFollowRedirects() method, 474
setFont() method, 281, 371
setHgap() method, 320
setIcon() method, 267

setIndentation() method, 541
setInt() method, 515
setJMenuBar() method, 305
setLayout() method
  card layouts, 326
  containers, 314
  panels, 325
setLineWrap() method, 269
setListData() method, 277
setLong() method, 515
setLookAndFeel() method
  dialog boxes, 293
  GUIs, 259
setMajorTickSpacing() method, 294
setMaximum() method, 301
set() method, 234
setMinimum() method, 301
setMinorTickSpacing() method, 294
setNull() method, 516
Set object, 494
setPaintLabels() method, 295
setPaint() method, 378
setPaintTicks() method, 295
setPreferredSize() method, 296
setRenderingHint() method, 372
setSelected() method, 272
setSelecteIndex() method, 275
setServerURL() method, 556
setShort() method, 515
setSize() method
  components, 264
  frames, 257
setSoTimeOut() method, 475

setString() method, 515
setStringPainted() method, 301
setStroke() method, 380
setText() method. 267-268
setting
  background color, 377
  breakpoints, 653-655
  drawing colors, 376-377
  system properties, 657
setValue() method, 301
setVgap() method, 320
setVisible() method
  components, 264
  frames, 258
setVisibleRowCount() method, 277
setWrapStyleWord() method, 269
shared behavior, 31-32
shared values, defining, 43
shell prompt. *See* command line
short data type, 42
showConfirmDialog() method, 287
showInputDialog() method, 288
showMessageDialog() method, 289
show() method, 326
showOptionDialog() method, 290
shrinking NetBeans panes, 604
signatures (digital), 404
signatures (methods), 128, 139
signing code, 658-659
Simple API for XML (SAX), 530
SimpleFrame.java, 260
Simple Object Access Protocol (SOAP), 551
@since tag (javadoc), 650

single inheritance, 31, 176
single-step execution, 652
SiteClient.java, 558
size() method
  array lists, 235
  elements, 537
  file channels, 489
  Map interface, 241
sizing
  components, 264
  frames, 257, 310
  scrolling panes, 296
slash character (/), XML tags, 401
Slider.java, 295
sliders, 294-296
  advantages, 294
  labels, 294-295
  orientation, 294
SOAP (Simple Object Access Protocol), 551
SocketChannel class, 493
Socket class, 475
SocketImpl class, 480
SocketImplFactory interface, 480
sockets, 475-479
  client-side
    closing, 476
    nonblocking clients, 493-499
    opening, 475
  opening, 493
  server-side, 479-480
    designing server applications, 480-482
    nonblocking servers, 493-499
    testing server applications, 482-483

timeout values, 475
solving problems. *See* troubleshooting
source code. *See also* listings
comments in, 646
converting, 641
writing in Android Studio, 584-591
source files
creating, 629
saving, 630
SourceReader.java, 439
sources (casting), 83
Spartacus.java, 601
specifying class files, 640
speed of Java programs, 642
splash screens, 407
SQLException, 509-511
SQL (Structured Query Language), 504-505
prepared statements, 514-516
queries, 508-511
square brackets ([]), arrays, 59, 96
sRGB color system, 375
Stack class, 227, 238-239
Stacker.java, 318
stack frames, 656
stacks, 227, 238-239, 656
Standard Widget Toolkit (SWT), 310
startActivity() method, 587
Start command (appletviewer), 643
start() method, 212
Statement interface, 510
statements, 38. *See also* commands; keywords; modifiers
block statements, 38, 103, 121
conditionals

if, 104-105
nested if, 106
switch, 105-111, 121
ternary operator, 112
empty, in loops, 114
expressions, 51-52
definition of, 38
operators. *See* operators
return values, 38, 52
finally, 199-202
import, 171-172, 175, 256
loops
breaking, 119
do, 118-119
for, 113-116
index values, 114
labeling, 120
restarting, 119-120
while, 116-118
package, 32, 63, 175
termination character, 38
static keyword, 24, 74, 126-127, 134
static methods, 164-167. *See also* class methods
static modifier, 158, 164
static variables, 75, 164-167
step command (jdb), 654
stop at command (jdb), 653
Stop command (appletviewer), 643
stop in command (jdb), 653
stopping threads, 217-218
Storefront application, 181-187
Storefront.java, 184
storefronts (online), creating, 181-187

storing
command-line arguments, 111
data, 427
streams, 419-420
buffers, 427-431, 484-486
byte buffers, 486
character sets, 487-488
byte streams, 420-422
file input streams, 422-425
file output streams, 425-427
channels, 488-491
nonblocking I/O network connections, 492-499
character streams, 420, 437
reading text files, 437-440
writing text files, 440-441
console input, 431-432
creating, 420
data streams, 433-436
exception handling, 421-422
filtering, 421, 427
input, 371
opening over Internet, 470-475
Path objects, 441-444
reading, 420
writing to, 421
string arithmetic, 60-61
StringChecker.java, 76
String class
selecting substrings, 201
valueOf() method, 79
string data type (XML-RPC), 550
string literals, 50-51
String objects

concatenation operator, 82
creating, 21
strings
in Android apps, 575-577
comparing, 88-89
concatenating, 60-61
converting to numbers, 86
dividing into tokens, 69-70
formatting, 77-78
handling, 82
length of, 91
switch statements, 121
strings.xml, 575-579
StringTokenizer class, 69
StringTokenizer objects, creating, 69-70
stringWidth() method, 373
strokes (drawing), 380-381
struct data type (XML-RPC), 550
Structured Query Language. *See* SQL
structures, data. *See* data structures
styles (fonts), selecting, 371
styles.xml, 579
subclasses, 25
casting objects, 84-85
defining, 26
final classes and, 168
method inheritance, 163
overriding methods, 148-149
protected versus default access, 162
variable scope, 132
Subscriptions.java, 277
subscripts (arrays), 98-99
substring() method, 77, 201

substrings, selecting, 201
subtraction operator, 52
superclasses, 25
  casting objects, 84-85
  indicating, 126
  methods in, calling, 150
  modifying, 27
  overriding methods, 148-149
  variable scope, 132
super keyword, 150
super() method, 151
surfaces for drawing, creating, 368-369
SurveyFrame.java, 333
SurveyWizard.java, 331
suspend command (jdb), 656
Swing, 255, 285, 310
  applications
    creating interface, 257-259
    developing framework, 260-261
    improving performance, 407-412
  components, 256, 264
    adding to containers, 256, 262-263
    AWT components versus, 256
    check boxes, 272-274
    combo boxes, 274-276
    creating, 256, 261-262
    dialog boxes, 286-293
    disabled, 264
    drop-down lists, 274-276
    hiding, 264
    image icons, 265-267
    labels, 267
    lists, 276-278

menus, 303-307
progress bars, 300-303
radio buttons, 272-274
resizing, 264
scrolling panes, 271-272, 296-297
sliders, 294-296
tabbed panes, 307-310
text areas, 269-271
text fields, 268
toolbars, 297-300
containers, panels, 262
event-handling, 339. *See also* event listeners
action events, 345-346
component setup, 341-342
focus events, 346-349
item events, 349-351
keyboard events, 351-352
methods, 342-345
mouse events, 352
mouse movement events, 352-357
window events, 357
Info application, 292
layout managers, 314-315
alternatives to, 334
border layout, 322-324
box layout, 317-319
card layout, 325-333
combining, 324-325
creating, 314
flow layout, 315-317
grid layout, 320-321
insets, 333-334
SwingConstants interface, 267, 294, 309

SwingWorker class, 407-413
switch statements, 105-111, 121
SWT (Standards Widget Toolkit), 310
synchronized modifier, 158
System class, 79, 657
  class methods, 134
  in variable (input stream), 431
System.out class variable, 79
System.out.format() method, 77-78
System.out.println() method, 46
System.out.print() method, 46
system properties, 656-658

T

tabbed panes, 307-310
tables in databases
  creating, 517
  viewing, 507-508
TabPanels.java, 308
Tag command (appletviewer), 644
tags
  javadoc, 647, 650
  XML, 401, 527-528
TCP sockets, 475-479
  client-side
    closing, 476
    opening, 475
  server-side, 479-480
    designing server applications, 480-482
    testing server applications, 482-483
telnet, running, 482
terminating. *See* stopping
termination character, 38

ternary operator, 59, 112
test variables, switch statements, 106
testing
  with loops, 113
  server applications, 482-483
  unit testing, 219
text, drawing, 370-372
  antialiasing, 372
  finding font information, 372-375
text areas, 255, 269-271
Text class, 533
text editors
  associating .java files with, 630
  selecting, 628-629
text fields, 255, 268
  event handling
    action events, 345-346
    item events, 349-351
  password fields, obscuring, 268
text files
  reading, 437-440
  writing, 440-441
TextFrame.java, 373
this keyword, 130-131, 145, 327
Thread class, 191, 211
threaded applications
  example, 213-217
  writing, 211-213
threads, 191, 211
  creating, 212
  running, 213
  stopping, 217-218
threads command (jdb), 656

Throwable class, 194-195
throwing exceptions, 194, 202, 207
  checked, 204
  inheritance issues, 206
  nested handlers, 208-209
  throws clause, 203
  unchecked, 204
throws keyword, 203-205
timeout values (sockets), 475
TimeServer.java, 481
TimeZone class, 657
TitleBar.java, 343
title tag, 401
T-Mobile G1, 570
toFile() method, 441
tokens, 69-70
TokenTester.java, 69
Tolksdorf, Robert, 640
toolbars, 297-300
tools
  development, selecting, 616
  functions, modifying, 638
toPath() method, 442
toUpperCase() method, 77
toXML() method, 534
Translations editor, 576
Transmission Control Protocol. *See* TCP sockets
transport-layer sockets, 480
trimToSize() method, 235
troubleshooting. *See also* debugging
  Android apps, 578, 610
    checking BIOS settings, 614
    installing HAXM, 611-613

arrays, 99
command-line arguments, 139
compiled classes, 601
compiling, 633
database connections, 514
errors, fatal, 196
for loops, 114
Java Development Kit (JDK) configuration, 624-628
methods, overloaded, 140
in NetBeans, 603
running Android apps, 591
variables
class variables, 75
scope, 131
true value (Boolean), 49
try and catch blocks, 196-199
finally clause, 199-202
Twitter, author contact information, 608

U

UIManager class, 259
UltraEdit, 629
unboxing, 87
unchecked exceptions, 194, 204
underscore (_), in large number literals, 48
Unicode character set, 40, 49-50, 420, 437, 487
Unicode Consortium website, 51
unit testing, 219
unsigned bytes, 434
up command (jdb), 655
updating
NetBeans, 598
progress bars, 301

URL (uniform resource locator), 471
URL class, 471
URL() constructor, 471
URL objects, creating, 471
USB drivers, installing, 591
user coordinate space, 378
user interface. *See* GUI
UTF-8 character set, 487
UTF-16 character set, 487
UTF-16BE character set, 487
UTF-16LE character set, 487
utilities
  appletviewer, 642-646
  command line, 638-639
  jar, 650-652
  jarsigner, 658-659
  javac compiler, 641-642
  javadoc, 646-650
  java interpreter, 639-641
  jdb debugger, 652-653
    advanced commands, 655-656
    applet debugging, 655
    application debugging, 653-655
  keytool, 658-659

V

validating XML, 529
valueOf() method, 79
values
  assigning to variables, 43
  class variables, modifying, 75
  of instance variables, modifying, 73-74
  passing arguments by, 132

shared values, defining, 43
variables
access control, 159
comparison of types, 163
default access, 159
private access, 159-161
protected access, 162
public access, 161
array variables, 96-97
assigning values, 40, 43
casting, definition of, 83
CLASSPATH
Windows 7-10, 633-635
Windows 98/Me, 635-636
class variables, 18, 39, 72, 127
accessing, 165
accessing values, 75
defining, 74
initial values, 40
instance variables versus, 33, 74
modifying values, 75
troubleshooting, 75
constant variables, 43-44
creating, 39-40, 44-45
declaring, 39-40
decrementing, 55-56
definition of, 38
encapsulation, 159
environment variables, 656
final variables, 167
incrementing, 55-56
in (input stream), 431
instance variables, 17, 39, 72

accessing values, 72-73
class variables versus, 33, 74
defining, 21, 126-127
initial values, 40
length, 99
modifying values, 73-74
nesting with method calls, 79
interface type, 178
in interfaces, 179
local variables, 39
naming, 40-41
overflow, 62
PATH
Windows 7-10, 625-627
Windows 98/Me, 627-628
protecting, 170
scope, 103, 131-132
lexical scope, 121
troubleshooting, 131
static, 164-167
test variables, switch statements, 106
types, 41
array elements, 98
casting, 82-84
class types, 43
converting to/from objects, 86-87
data types, 42-43
objects versus, 91
Variables.java, 45
Vector class, 235
vendor tag, 401
verbose compiler, 642
version numbers

Android, 592
Java Development Kit (JDK), 624
@version tag (javadoc), 647
viewing
database connection information, 510
documents (HTML), 643
Java documentation, 47
tabbed panes, 307
tables in databases, 507-508
visible frames, 258
void data type, 43
void return type (methods), 129
volatile modifier, 158

W

Weather.java, 53
web-launched applications (Java Web Start), 392-395
configuring web servers for, 405
creating JNLP files, 396-404
description tag, 406
icon tag, 406-407
security, 405-406
WebReader.java, 471
WebServer class, 559
WebServer() constructor, 559
web services, 549. *See also* XML-RPC
well-formed XML, 528
while loops, 116-118
white space, adding to XML documents, 540-542
windowActivated() method, 357
WindowAdapter class, 358, 456
windowClosed() method, 357
windowClosing() method, 357

windowDeactivated() method, 357
windowDeiconified() method, 357
window events, 340, 357
windowIconified() method, 357
WindowListener event listener, 340, 357
WindowListener interface, 455
windowOpened() method, 357
windows
  absolute component placement, 334
  frames
    closing, 259-260
    developing framework, 260-261
    displaying, 258
    locations, 258
    sizing, 257
    visible, 258
  layout managers. *See* layout managers
Windows
  command-line interface, 619-621
    creating folders, 622-623
    opening folders, 621-622
    running programs, 623
  installing Java Development Kit (JDK), 617-619
  Java programs
    compiling in, 631-632
    running in, 631-632
Windows 7-10
  CLASSPATH variable, 633-635
  PATH variable, 625-627
Windows 98/Me
  CLASSPATH variable, 635-636
  PATH variable, 627-628
wizard interfaces, 327

Word, 629
WordPad, 628
word processors. *See* text editors
workbench.rss, 526
wrap() method, 484
wrapper classes, 134
writeBoolean() method, 433
writeByte() method, 433
writeDouble() method, 433
writeFloat() method, 433
writeInt() method, 433
writeLong() method, 433
write() method
  buffered character streams, 441
  buffered output streams, 428
  character streams, 440
  char versus int data types, 445
  file output streams, 425
  filters, 421
  streams, 421
  XML documents, 541
Writer class, 437
writeShort() method, 433
writing
  applications, threaded, 211-217
  apps in Android Studio, 575-577
  buffered character streams, 440
  to buffered output streams, 428
  bytes, multiple, 425
  database records, 514-521
  data output streams, 433
  Java code in Android Studio, 584-591
  to streams, 421

text files, 440-441

X–Z

XML (Extensible Markup Language), 525
advantages, 526
dialects, designing, 528-529
documents
creating, 532-535
formatting, 540-542
modifying, 536-540
files, editing, 576-577
processing
evaluating XOM, 542-545
with Java, 530
with XOM, 530-532
reason for name, 546
RSS and, 526
tags, 401, 527-528
validating, 529
well-formed XML, 528
XML Object Model. *See* XOM
XML-RPC, 549-551
Apache XML-RPC
data types supported, 565
installing, 554-556
clients, 556-559
data types supported, 550
debuggers, 554
requests
responding to, 553-554
sending, 551-552
servers, 559-564
XmlRpcClient class, 556

XmlRpcServer class, 559
?xml tag, 527
XOM (XML Object Model), 530-532
  adding to NetBeans, 532
  creating XML documents, 532-535
  evaluating, 542-545
  formatting XML documents, 540-542
  licensing, 531
  modifying XML documents, 536-540
  principles, 531
XOR operator, 58
XYZ color system, 375

zero-based methods, 77

Learning Labs!

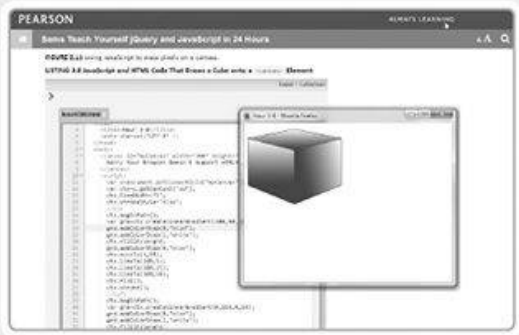
Learn online with videos, live code editing, and quizzes

Learning Labs are interactive, self-paced courses designed to teach learners of all types.

Visit informit.com/learninglabs to see available labs, tryout full samples, and order today.

- Read the complete text of the book online in your web browser

- Watch an expert instructor show you how to perform tasks in easy-to-follow videos



- Try your hand at coding in an interactive code-editing sandbox in select products

- Test yourself with interactive quizzes

ALWAYS LEARNING PEARSON

REGISTER YOUR PRODUCT at informit.com/register

Access Additional Benefits and SAVE 35% on Your Next Purchase

- Download available product updates.
- Access bonus material when applicable.
- Receive exclusive offers on new editions and related products. (Just check the box to hear from us when setting up your account.)
- Get a coupon for 35% for your next purchase, valid for 30 days. Your code will be available in your InformIT cart. (You will also find it in the Manage Codes section of your account page.)

Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

InformIT.com–The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com you can

- Shop our books, eBooks, software, and video training.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletters (informit.com/newsletters).
- Read free articles and blogs by information technology experts.
- Access thousands of free chapters and video lessons.

Connect with InformIT–Visit informit.com/community

Learn about InformIT community events and programs.

informIT.com

the trusted technology learning source

Addison-Wesley • Cisco Press • IBM Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • VMware Press

Code Snippets

```
Monospace looks like this. Hi, mom!
```

```
 1: class MarsRobot {
 2:     String status;
 3:     int speed;
 4:     float temperature;
 5:
 6:     void checkTemperature() {
 7:         if (temperature < -80) {
 8:             status = "returning home";
 9:             speed = 5;
10:         }
11:     }
12:
13:     void showAttributes() {
14:         System.out.println("Status: " + status);
15:         System.out.println("Speed: " + speed);
16:         System.out.println("Temperature: " + temperature);
17:     }
18: }
```

```
void checkTemperature() {
    if (temperature < -80) {
        status = "returning home";
        speed = 5;
    }
}
```

```
void showAttributes() {
    System.out.println("Status: " + status);
    System.out.println("Speed: " + speed);
    System.out.println("Temperature: " + temperature);
}
```

```
 1: class MarsApplication {
 2:     public static void main(String[] arguments) {
 3:         MarsRobot spirit = new MarsRobot();
 4:         spirit.status = "exploring";
 5:         spirit.speed = 2;
 6:         spirit.temperature = -60;
 7:
 8:         spirit.showAttributes();
 9:         System.out.println("Increasing speed to 3.");
10:         spirit.speed = 3;
11:         spirit.showAttributes();
12:         System.out.println("Changing temperature to -90.");
13:         spirit.temperature = -90;
14:         spirit.showAttributes();
15:         System.out.println("Checking the temperature.");
16:         spirit.checkTemperature();
17:         spirit.showAttributes();
18:     }
19: }
```

```
int weight = 225;
System.out.println("Free the bound periodicals!");
song.duration = 230;
```

```
spirit.speed = 2; spirit.temperature = -60;
```

```
public static void main(String[] arguments) {
    int total;
    String reportTitle;
    boolean active;
}
```

```
String zipCode = "02134";

int box = 350;
boolean pbs = true;
String name = "Zoom", city = "Boston", state = "MA";
```

```
 1: package com.java21days;
 2:
 3: public class Variables {
 4:
 5:     public static void main(String[] arguments) {
 6:         final char UP = 'U';
 7:         byte initialLevel = 12;
 8:         short location = 13250;
 9:         int score = 3500100;
10:         boolean newGame = true;
11:
12:         System.out.println("Level: " + initialLevel);
13:         System.out.println("Up: " + UP);
14:     }
15: }
```

```
int creditHours = 3; // set up credit hours for course
```

```
/* This program occasionally deletes all files on
your hard drive and renders it unusable
forever when you click the Save button. */
```

```
String quitMsg = "Are you sure you want to quit?";

String password = "drowssap";
```

```
String example = "Socrates asked, \"Hemlock is poison?\"";
System.out.println("Sincerely,\nMillard Fillmore\n");
String title = "Sams Teach Yourself Node in the John\u2122";
```

```
 1: package com.java21days;
 2:
 3: public class Weather {
 4:     public static void main(String[] arguments) {
 5:         float fah = 86;
 6:         System.out.println(fah + " degrees Fahrenheit is ...");
 7:         // To convert Fahrenheit into Celsius
 8:         // begin by subtracting 32
 9:         fah = fah - 32;
10:         // Divide the answer by 9
11:         fah = fah / 9;
12:         // Multiply that answer by 5
13:         fah = fah * 5;
14:         System.out.println(fah + " degrees Celsius\n");
15:
16:         float cel = 33;
17:         System.out.println(cel + " degrees Celsius is ...");
18:         // To convert Celsius into Fahrenheit
19:         // begin by multiplying by 9
20:         cel = cel * 9;
21:         // Divide the answer by 5
22:         cel = cel / 5;
23:         // Add 32 to the answer
24:         cel = cel + 32;
25:         System.out.println(cel + " degrees Fahrenheit");
26:     }
27: }
```

```
int x, y, z; // x, y, and z are declared
x = 42;      // x is given the value 42
y = x++;     // y is given x's value (42) before it is incremented
             // and x is then incremented to 43
z = ++x;     // x is incremented to 44, and z is given x's value
```

```
boolean extraLife = (score > 75000) & (playerLives < 10);
```

```
boolean extralife = (score > 75000) || (playerLevel == 0);
```

```
String brand = "Jif";
System.out.println("Choosy mothers choose " + brand);
```

```
System.out.println(4 + " score and " + 7 + " years ago");
```

```
String name = new String("Hal Jordan");

URL address = new URL("http://www.java21days.com");

MarsRobot robbie = new MarsRobot();
```

```
Random seed = new Random(606843071);

Point pt = new Point(0, 0);
```

```
 1: package com.java21days;
 2:
 3: import java.util.StringTokenizer;
 4:
 5: class TokenTester {
 6:
 7:     public static void main(String[] arguments) {
 8:         StringTokenizer st1, st2;
 9:
10:         String quote1 = "GOOG 530.80 -9.98";
11:         st1 = new StringTokenizer(quote1);
12:         System.out.println("Token 1: " + st1.nextToken());
13:         System.out.println("Token 2: " + st1.nextToken());
14:         System.out.println("Token 3: " + st1.nextToken());
15:
16:         String quote2 = "RHT@75.00@0.22";
17:         st2 = new StringTokenizer(quote2, "@");
18:         System.out.println("\nToken 1: " + st2.nextToken());
19:         System.out.println("Token 2: " + st2.nextToken());
20:         System.out.println("Token 3: " + st2.nextToken());
21:     }
22: }
```

```
float total = customer.orderTotal;
```

```
float total = store.customer.orderTotal;
```

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class PointSetter {
 6:
 7:     public static void main(String[] arguments) {
 8:         Point location = new Point(4, 13);
 9:
10:         System.out.println("Starting location:");
11:         System.out.println("X equals " + location.x);
12:         System.out.println("Y equals " + location.y);
13:
14:         System.out.println("\nMoving to (7, 6)");
15:         location.x = 7;
16:         location.y = 6;
17:
18:         System.out.println("\nEnding location:");
19:         System.out.println("X equals " + location.x);
20:         System.out.println("Y equals " + location.y);
21:     }
22: }
```

```
class FamilyMember {
    static String surname = "Mendoza";
    String name;
    int age;
}
```

```
FamilyMember dad = new FamilyMember();
System.out.println("Family's surname is: " + dad.surname);
System.out.println("Family's surname is: " + FamilyMember.surname);
```

```
customer.addToCart(itemNumber, price, quantity);
```

```
 1: package com.java21days;
 2:
 3: class StringChecker {
 4:
 5:     public static void main(String[] arguments) {
 6:         String str = "A Lannister always pays his debts";
 7:         System.out.println("The string is: " + str);
 8:         System.out.println("Length of this string: "
 9:             + str.length());
10:         System.out.println("The character at position 6: "
11:             + str.charAt(6));
12:         System.out.println("The substring from 12 to 18: "
13:             + str.substring(12, 18));
14:         System.out.println("The index of the first 't': "
15:             + str.indexOf('t'));
16:         System.out.println("The index of the beginning of the "
17:             + "substring \"debts\": " + str.indexOf("debts"));
18:         System.out.println("The string in uppercase: "
19:             + str.toUpperCase());
20:     }
21: }
```

```
int accountBalance = 5005;
System.out.format("Balance: $%,d%n", accountBalance);
```

```
double pi = Math.PI;
System.out.format("%.11f%n", pi);
```

```
String label = "From";
String upper = label.toUpperCase();
```

```
customer.cancelOrder().fileComplaint();
```

```
customer.orderTotal.putOnLayaway(itemNumber, price, quantity);
```

```
int firstPrice = 225;
int secondPrice = 217;
int higherPrice = Math.max(firstPrice, secondPrice);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class RefTester {
 6:     public static void main(String[] arguments) {
 7:         Point pt1, pt2;
 8:         pt1 = new Point(100, 100);
 9:         pt2 = pt1;
10:
11:         pt1.x = 200;
12:         pt1.y = 200;
13:         System.out.println("Point1: " + pt1.x + ", " + pt1.y);
14:         System.out.println("Point2: " + pt2.x + ", " + pt2.y);
15:     }
16: }
```

```
float gpa = 2.25F;
System.out.println("Honest, mom, my GPA is a " +
(gpa + 1.5));
```

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep; // no cast needed for upward use
veep = (VicePresident) emp; // must cast explicitly
```

```
Graphics2D screen2D = (Graphics2D) screen;
```

```
Integer dataCount = new Integer(7801);
```

```
int newCount = dataCount.intValue(); // returns 7801
```

```
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

```
Float f1 = 12.5F;
Float f2 = 27.2F;
System.out.println("Lower number: " + Math.min(f1, f2));
```

```
 1: package com.java21days;
 2:
 3: class EqualsTester {
 4:     public static void main(String[] arguments) {
 5:         String str1, str2;
 6:         str1 = "Boy, that escalated quickly.";
 7:         str2 = str1;
 8:
 9:         System.out.println("String1: " + str1);
10:         System.out.println("String2: " + str2);
11:         System.out.println("Same object? " + (str1 == str2));
12:
13:         str2 = new String(str1);
14:
15:         System.out.println("String1: " + str1);
16:         System.out.println("String2: " + str2);
17:         System.out.println("Same object? " + (str1 == str2));
18:         System.out.println("Same value? " + str1.equals(str2));
19:     }
20: }
```

```
String name = key.getClass().getName();
```

```java
boolean check1 = "Texas" instanceof String; // true

Object obiwan = new Object();
boolean check2 = obiwan instanceof String; // false
```

```
public class AyeAye {
     int i = 40;
     int j;

     public AyeAye() {
          setValue(i++);
     }

     void setValue(int inputValue) {
          int i = 20;
          j = i + 1;
          System.out.println("j = " + j);
     }
}
```

```
String[] players = new String[10];
```

```
Integer[] series = new Integer[3];
series[0] = new Integer(10);
series[1] = new Integer(3);
series[2] = new Integer(5);
```

```
Point[] markup = { new Point(1,5), new Point(3,3), new Point(2,3) };
```

```
String[] titles = { "Mr.", "Mrs.", "Ms.", "Miss", "Dr." };
```

```
float[] rating = new float[20];
rating[20] = 3.22F;
```

```
System.out.println("Elements: " + rating.length);
```

```
 1: package com.java21days;
 2:
 3: class HalfDollars {
 4:     public static void main(String[] arguments) {
 5:         int[] denver = { 1_700_000, 4_600_000, 2_100_000 };
 6:         int[] philadelphia = new int[denver.length];
 7:         int[] total = new int[denver.length];
 8:         int average;
 9:
10:         philadelphia[0] = 1_800_000;
11:         philadelphia[1] = 5_000_000;
12:         philadelphia[2] = 2_500_000;
13:
14:         total[0] = denver[0] + philadelphia[0];
15:         total[1] = denver[1] + philadelphia[1];
16:         total[2] = denver[2] + philadelphia[2];
17:         average = (total[0] + total[1] + total[2]) / 3;
18:
19:         System.out.print("2012 production: ");
20:         System.out.format("%,d%n", total[0]);
21:         System.out.print("2013 production: ");
22:         System.out.format("%,d%n", total[1]);
23:         System.out.print("2014 production: ");
24:         System.out.format("%,d%n", total[2]);
25:         System.out.print("Average production: ");
26:         System.out.format("%,d%n", average);
27:     }
28: }
```

```
int[][] dayValue = new int[53][7];
```

```
int[][][] cen = new int[100][52][7];
System.out.println("Elements in 1st dimension: " + cen.length);
System.out.println("Elements in 2nd dimension: " + cen[0].length);
System.out.println("Elements in 3rd dimension: " + cen[0][0].length);
```

```
if (arguments.length < 3) {
    System.out.println("Not enough arguments");
    System.exit(-1);
}
```

```
if (operation == '+')
     add(object1, object2);
else if (operation == '-')
     subtract(object1, object2);
else if (operation == '*')
     multiply(object1, object2);
else if (operation == '/')
    divide(object1, object2);
```

```
char grade = 'D';
switch (grade) {
    case 'A':
        System.out.println("Great job!");
        break;
    case 'B':
        System.out.println("Good job!");
        break;
    case 'C':
        System.out.println("You can do better!");
        break;
    default:
        System.out.println("Consider cheating!");
}
```

```
String command = "close";
switch (command) {
     case "open":
          openFile();
          break;
     case "close":
          closeFile();
          break;
     default:
          System.out.println("Invalid command");
     }
}
```

```
switch (operation) {
    case '+':
        add(object1, object2);
        break;
    case '-':
        subtract(object1, object2);
        break;
    case '*':
        multiply(object1, object2);
        break;
    case '/':
        divide(object1, object2);
        break;
}
```

```java
int x = 5;
switch (x) {
    case 2:
    case 4:
    case 6:
    case 8:
        System.out.println("x is an even number");
        break;
    default:
        System.out.println("x is an odd number");
}
```

```
 1: package com.java21days;
 2:
 3: class DayCounter {
 4:     public static void main(String[] arguments) {
 5:         int yearIn = 2016;
 6:         int monthIn = 1;
 7:         if (arguments.length > 0)
 8:             monthIn = Integer.parseInt(arguments[0]);
 9:         if (arguments.length > 1)
10:             yearIn = Integer.parseInt(arguments[1]);
11:         System.out.println(monthIn + "/" + yearIn + " has "
12:             + countDays(monthIn, yearIn) + " days.");
13:     }
14:
15:     static int countDays(int month, int year) {
16:         int count = -1;
17:         switch (month) {
18:             case 1:
19:             case 3:
20:             case 5:
21:             case 7:
22:             case 8:
23:             case 10:
24:             case 12:
```

```
25:                     count = 31;
26:                     break;
27:                 case 4:
28:                 case 6:
29:                 case 9:
30:                 case 11:
31:                     count = 30;
32:                     break;
33:                 case 2:
34:                     if (year % 4 == 0)
35:                         count = 29;
36:                     else
37:                         count = 28;
38:                     if ((year % 100 == 0) & (year % 400 != 0))
39:                         count = 28;
40:             }
41:             return count;
42:         }
43: }
```

```
test ? trueResult : falseResult;
```

```
int ourBestScore = myScore > yourScore ? myScore : yourScore;
```

```
for (initialization; test; increment) {
    statement;
}
```

```
String[] salutation = new String[10];
int i; // the loop index variable
for (i = 0; i < salutation.length; i++) {
    salutation[i] = "Mr.";
}
```

```
for (i = 4001; notPrime(i); i += 2);
```

```
int x = 1;
for (i = 0; i < 10; i++);
    x = x * i; // this line is not inside the loop!
```

```
 1: package com.java21days;
 2:
 3: class HalfLooper {
 4:     public static void main(String[] arguments) {
 5:         int[] denver = { 1_700_000, 4_600_000, 2_100_000 };
 6:         int[] philadelphia = { 1_800_000, 5_000_000, 2_500_000 };
 7:         int[] total = new int[denver.length];
 8:         int sum = 0;
 9:
10:         for (int i = 0; i < denver.length; i++) {
11:             total[i] = denver[i] + philadelphia[i];
12:             System.out.format((i + 2012) + " production: %,d%n",
13:                 total[i]);
14:             sum += total[i];
15:         }
16:
17:         System.out.format("Average production: %,d%n",
18:             (sum / denver.length));
19:     }
20: }
```

```
while (i < 13) {
    x = x * i++; // the body of the loop
}
```

```
 1: package com.java21days;
 2:
 3: class ArrayCopier {
 4:     public static void main(String[] arguments) {
 5:         int[] array1 = { 7, 4, 8, 1, 4, 1, 4 };
 6:         float[] array2 = new float[array1.length];
 7:
 8:         System.out.print("array1: [ ");
 9:         for (int i = 0; i < array1.length; i++) {
10:             System.out.print(array1[i] + " ");
11:         }
12:         System.out.println("]");
13:
14:         System.out.print("array2: [ ");
15:         int count = 0;
16:         while ( count < array1.length && array1[count] != 1) {
17:             array2[count] = (float) array1[count];
18:             System.out.print(array2[count++] + " ");
19:         }
20:         System.out.println("]");
21:     }
22: }
```

```
long i = 1;
do {
     i *= 2;
     System.out.print(i + " ");
} while (i < 3_000_000_000_000L);
```

```
int count = 0;
while (count < array1.length) {
     if (array1[count] == 1) {
          break;
     }
     array2[count] = (float) array2[count++];
}
```

```
int count = 0;
int count2 = 0;
while (count++ <= array1.length) {
    if (array1[count] == 1) {
        continue;
    }
    array2[count2++] = (float) array1[count];
}
```

```
out: for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 50; j++) {
        if (i * j > 400) {
            break out;
        }
    }
}
```

```
public class Cases {
    public static void main(String[] arguments) {
        float x = 9;
        float y = 5;
        int z = (int)(x / y);
        switch (z) {
            case 1:
                x = x + 2;
            case 2:
                x = x + 3;
            default:
                x = x + 1;
        }
        System.out.println("Value of x: " + x);
    }
}
```

```
class SportsTicker extends Ticker {
    // body of the class
}
```

```
class MarsRobot extends ScienceRobot {
    String status;
    int speed;
    float temperature;
    int power;
}
```

```
static int SUM;
static final int MAX_OBJECTS = 10;
```

```
returnType methodName(type1 arg1, type2 arg2, type3 arg3 ...) {
    // body of method
}
```

```
int[] makeRange(int lower, int upper) {
    // body of method
}
```

```
 1: package com.java21days;
 2:
 3: class RangeLister {
 4:     int[] makeRange(int lower, int upper) {
 5:         int[] range = new int[(upper-lower) + 1];
 6:
 7:         for (int i = 0; i < range.length; i++) {
 8:             range[i] = lower++;
 9:         }
10:         return range;
11:     }
12:
13:     public static void main(String[] arguments) {
14:         int[] range;
15:         RangeLister lister = new RangeLister();
16:
17:         range = lister.makeRange(4, 13);
18:         System.out.print("The array: [ ");
19:         for (int i = 0; i < range.length; i++) {
20:             System.out.print(range[i] + " ");
21:         }
22:         System.out.println("]");
23:     }
24:
25: }
```

```
t = this.x;              // the x instance variable for this object

z.resetData(this);       // call the resetData method, defined in
                         // the z class, and pass it the current object
return this;             // return the current object
```

```
t = x;                  // the x instance variable for this object
```

```
class ScopeTest {
    int test = 10;

    void printTest() {
        int test = 20;
        System.out.println("Test: " + test);
    }

    public static void main(String[] arguments) {
        ScopeTest st = new ScopeTest();
        st.printTest();
    }
}
```

```
 1: package com.java21days;
 2:
 3: class Passer {
 4:
 5:     void toUpperCase(String[] text) {
 6:         for (int i = 0; i < text.length; i++) {
 7:             text[i] = text[i].toUpperCase();
 8:         }
 9:     }
10:
11:     public static void main(String[] arguments) {
12:         Passer passer = new Passer();
13:         passer.toUpperCase(arguments);
14:         for (int i = 0; i < arguments.length; i++) {
15:             System.out.print(arguments[i] + " ");
16:         }
17:         System.out.println();
18:     }
19: }
```

```
System.exit(0);

long now = System.currentTimeMillis();
```

```
static void exit(int argument) {
    // body of method
}
```

```
int count = Integer.parseInt("42");
```

```
public static void main(String[] arguments) {
    // body of method
}
```

```
java Echo Wilhelm Niekro Hough "Tim Wakefield" 49
```

```
public static void main(String[] arguments) {
    // body of method
}
```

```
 1: package com.java21days;
 2:
 3: class Averager {
 4:     public static void main(String[] arguments) {
 5:         int sum = 0;
 6:
 7:         if (arguments.length > 0) {
 8:             for (int i = 0; i < arguments.length; i++) {
 9:                 sum += Integer.parseInt(arguments[i]);
10:             }
11:             System.out.println("Sum is: " + sum);
12:             System.out.println("Average is: " +
13:                 (float) sum / arguments.length);
14:         }
15:     }
16: }
```

```
Box buildBox(int x1, int y1, int x2, int y2) {
    this.x1 = x1;
    this.y1 = y1;
    this.x2 = x2;
    this.y2 = y2;
    return this;
}
```

```
Box buildBox(Point topLeft, Point bottomRight) {
    x1 = topLeft.x;
    y1 = topLeft.y;
    x2 = bottomRight.x;
    y2 = bottomRight.y;
    return this;
}
```

```
Box buildBox(Point topLeft, int w, int h) {
    x1 = topLeft.x;
    y1 = topLeft.y;
    x2 = (x1 + w);
    y2 = (y1 + h);
    return this;
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class Box {
 6:     int x1 = 0;
 7:     int y1 = 0;
 8:     int x2 = 0;
 9:     int y2 = 0;
10:
11:     Box buildBox(int x1, int y1, int x2, int y2) {
12:         this.x1 = x1;
13:         this.y1 = y1;
14:         this.x2 = x2;
15:         this.y2 = y2;
16:         return this;
17:     }
18:
19:     Box buildBox(Point topLeft, Point bottomRight) {
20:         x1 = topLeft.x;
21:         y1 = topLeft.y;
22:         x2 = bottomRight.x;
23:         y2 = bottomRight.y;
24:         return this;
25:     }
26:
```

```
27:     Box buildBox(Point topLeft, int w, int h) {
28:         x1 = topLeft.x;
29:         y1 = topLeft.y;
30:         x2 = (x1 + w);
31:         y2 = (y1 + h);
32:         return this;
33:     }
34:
35:     void printBox(){
36:         System.out.print("Box: <" + x1 + ", " + y1);
37:         System.out.println(", " + x2 + ", " + y2 + ">");
38:     }
39:
40:     public static void main(String[] arguments) {
41:         Box rect = new Box();
42:
43:         System.out.println("Calling buildBox with "
44:             + "coordinates (25,25) and (50,50):");
45:         rect.buildBox(25, 25, 50, 50);
46:         rect.printBox();
47:
48:         System.out.println("\nCalling buildBox with "
49:             + "points (10,10) and (20,20):");
50:         rect.buildBox(new Point(10, 10), new Point(20, 20));
51:         rect.printBox();
```

```
52:
53:         System.out.println("\nCalling buildBox with "
54:             + "point (10,10), width 50 and height 50:");
55:
56:         rect.buildBox(new Point(10, 10), 50, 50);
57:         rect.printBox();
58:     }
59: }
```

```
Box buildBox(Point topLeft, Point bottomRight) {
    return buildBox(topLeft.x, topLeft.y,
        bottomRight.x, bottomRight.y);
}
```

```
rect.buildBox(new Point(10, 10), 50, 50);
```

```
Point rectangle = new Point(10, 10), 50, 50);
rect.buildBox(new Point(10, 10), 50, 50);
```

```
class MarsRobot {
    String status;
    int speed;
    int power;

    MarsRobot(String in1, int in2, int in3) {
        status = in1;
        speed = in2;
        power = in3;
    }
}
```

```
MarsRobot curiosity = new MarsRobot("exploring", 5, 200);
```

```
this(argument1, argument2, argument3);
```

```
class Circle {
    int x, y, radius;

    Circle(int xPoint, int yPoint, int radiusLength) {
        this.x = xPoint;
        this.y = yPoint;
        this.radius = radiusLength;
    }

    Circle(int xPoint, int yPoint) {
        this(xPoint, yPoint, 1);
    }
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class Box2 {
 6:     int x1 = 0;
 7:     int y1 = 0;
 8:     int x2 = 0;
 9:     int y2 = 0;
10:
11:     Box2(int x1, int y1, int x2, int y2) {
12:         this.x1 = x1;
13:         this.y1 = y1;
14:         this.x2 = x2;
15:         this.y2 = y2;
16:     }
17:
18:     Box2(Point topLeft, Point bottomRight) {
19:         this(topLeft.x, topLeft.y, bottomRight.x,
20:             bottomRight.y);
21:     }
22:
23:     Box2(Point topLeft, int w, int h) {
24:         this(topLeft.x, topLeft.y, topLeft.x + w,
25:             topLeft.y + h);
```

```
26:     }
27:
28:     void printBox() {
29:         System.out.print("Box: <" + x1 + ", " + y1);
30:         System.out.println(", " + x2 + ", " + y2 + ">");
31:     }
32:
33:     public static void main(String[] arguments) {
34:         Box2 rect;
35:
36:         System.out.println("Calling Box2 with coordinates "
37:             + "(25,25) and (50,50):");
38:         rect = new Box2(25, 25, 50, 50);
39:         rect.printBox();
40:
41:         System.out.println("\nCalling Box2 with points "
42:             + "(10,10) and (20,20):");
43:         rect = new Box2(new Point(10, 10), new Point(20, 20));
44:         rect.printBox();
45:
46:         System.out.println("\nCalling Box2 with 1 point "
47:             + "(10,10), width 50 and height 50:");
48:         rect = new Box2(new Point(10, 10), 50, 50);
49:         rect.printBox();
50:
51:     }
52: }
```

```
 1: package com.java21days;
 2:
 3: class Printer {
 4:     int x = 0;
 5:     int y = 1;
 6:
 7:     void printMe() {
 8:         System.out.println("x is " + x + ", y is " + y);
 9:         System.out.println("I am an instance of the class " +
10:             this.getClass().getName());
11:     }
12: }
13:
14: class SubPrinter extends Printer {
15:     int z = 3;
16:
17:     public static void main(String[] arguments) {
18:         SubPrinter obj = new SubPrinter();
19:         obj.printMe();
20:     }
21: }
```

```
void printMe() {
    System.out.println("x is " + x + ", y is " + y +
        ", z is " + z);
    System.out.println("I am an instance of the class " +
        this.getClass().getName());
}
```

```
void doMethod(String a, String b) {
    // do stuff here
    super.doMethod(a, b);
    // do more stuff here
}
```

```
super(argument1, argument2, ...);
```

```
if (condition == true) {
    super(1,2,3); // call one superclass constructor
} else {
    super(1,2); // call a different constructor
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.Point;
 4:
 5: class NamedPoint extends Point {
 6:     String name;
 7:
 8:     NamedPoint(int x, int y, String name) {
 9:         super(x, y);
10:         this.name = name;
11:     }
12:
13:     public static void main(String[] arguments) {
14:         NamedPoint np = new NamedPoint(5, 5, "SmallPoint");
15:         System.out.println("x is " + np.x);
16:         System.out.println("y is " + np.y);
17:         System.out.println("Name is " + np.name);
18:     }
19: }
```

```
int total(int arg1, int arg2, int arg3) { ... }
float total(int arg1, int arg2, int arg3) { ... }
```

```java
public class BigValue {
    float result;

    public BigValue(int a, int b) {
        result = calculateResult(a, b);
    }

    float calculateResult(int a, int b) {
        return (a * 10) + (b * 2);
    }

    public static void main(String[] arguments) {
        BiggerValue bgr = new BiggerValue(2, 3, 4);
        System.out.println("The result is " + bgr.result);
    }
}

class BiggerValue extends BigValue {

    BiggerValue(int a, int b, int c) {
        super(a, b);
        result = calculateResult(a, b, c);
    }

    // answer goes here
        return (c * 3) * result;
    }
}
```

```
public class RedButton extends javax.swing.JButton {
    // ...
}

private boolean offline;

static final double WEEKS = 9.5;

protected static final int MEANING_OF_LIFE = 42;

public static void main(String[] arguments) {
    // body of method
}
```

```
class Logger {
    private String format;

    public String getFormat() {
        return this.format;
    }

    public void setFormat(String fmt) {
        if ( (fmt.equals("common")) || (fmt.equals("combined")) ) {
            this.format = fmt;
        }
    }
}
```

```
if (yard < 0) {
    System.out.println("Touchdown!");
    score = score + Football.TOUCHDOWN;
}
```

```
public static void main(String[] arguments) {
    // ...
}
```

```
protected boolean outOfData = true;
```

```
class AudioPlayer {

    private boolean openSpeaker(Speaker sp) {
        // implementation here
    }
}
```

```java
public class Circle {
    public static double PI = 3.14159265F;
    public double radius;

    public double area() {
        return PI * radius * radius;
    }
}
```

```
double circumference = 2 * Circle.PI * radius;
double randomNumber = Math.random();
```

```
 1: package com.java21days;
 2:
 3: public class InstanceCounter {
 4:     private static int numInstances = 0;
 5:
 6:     protected static int getCount() {
 7:         return numInstances;
 8:     }
 9:
10:     private static void addInstance() {
11:         numInstances++;
12:     }
13:
14:     InstanceCounter() {
15:         InstanceCounter.addInstance();
16:     }
17:
18:     public static void main(String[] arguments) {
19:         System.out.println("Starting with " +
20:             InstanceCounter.getCount() + " objects");
21:         for (int  i = 0; i < 500; ++i) {
22:             new InstanceCounter();
23:         }
24:         System.out.println("Created " +
25:             InstanceCounter.getCount() + " objects");
26:     }
27: }
```

```
public static final int TOUCHDOWN = 6;

static final String TITLE = "Captain";
```

```
public final void getSignature() {
    // body of method
}
```

```
public final class ChatServer {
    // body of method
}
```

```
public abstract class Palette {
    // ...
}
```

```
java.awt.Font text = new java.awt.Font();
```

```
import static java.lang.Math.*;
```

```java
import static java.lang.Math.*;

public class ShortConstants {
    public static void main(String[] arguments) {
        System.out.println("PI: " + PI);
        System.out.println("" + (PI * 3));
    }
}
```

```
java.util.Date = new java.util.Date();
```

```
public class AnimatedSign extends Sign
    implements Runnable {
    //...
}
```

```
public class AnimatedSign extends Sign
    implements Runnable, Observer {

    // ...
}
```

```
public interface Expandable {
    public abstract void expand(); // explicitly public and abstract
    void contract(); // effectively public and abstract
}
```

```java
public interface Expandable {
    public static final int INCREMENT = 10;
    long CAPACITY = 15000; // becomes public static and final

    public abstract void expand(); // explicitly public and abstract
    void contract(); // effectively public and abstract
}
```

```
public interface Trackable {
    public abstract Trackable beginTracking(Trackable self);
}
```

```
public class Monitor implements Trackable {

public Trackable beginTracking(Trackable self) {
    Monitor mon = (Monitor) self;
    // ...
    return mon;
}
```

```
interface PreciselyTrackable extends Trackable {
    // ...
}
```

```
 1: package org.cadenhead.ecommerce;
 2:
 3: public class Item implements Comparable {
 4:     private String id;
 5:     private String name;
 6:     private double retail;
 7:     private int quantity;
 8:     private double price;
 9:
10:     Item(String idIn, String nameIn, String retailIn, String qIn) {
11:         id = idIn;
12:         name = nameIn;
13:         retail = Double.parseDouble(retailIn);
14:         quantity = Integer.parseInt(qIn);
15:
16:         if (quantity > 400)
17:             price = retail * .5D;
18:         else if (quantity > 200)
19:             price = retail * .6D;
20:         else
21:             price = retail * .7D;
22:         price = Math.floor( price * 100 + .5 ) / 100;
23:     }
24:
25:     public int compareTo(Object obj) {
26:         Item temp = (Item) obj;
27:         if (this.price < temp.price) {
28:             return 1;
29:         } else if (this.price > temp.price) {
```

```
30:             return -1;
31:         }
32:         return 0;
33:     }
34:
35:     public String getId() {
36:         return id;
37:     }
38:
39:     public String getName() {
40:         return name;
41:     }
42:
43:     public double getRetail() {
44:         return retail;
45:     }
46:
47:     public int getQuantity() {
48:         return quantity;
49:     }
50:
51:     public double getPrice() {
52:         return price;
53:     }
54: }
```

```
 1: package org.cadenhead.ecommerce;
 2:
 3: import java.util.*;
 4:
 5: public class Storefront {
 6:     private LinkedList catalog = new LinkedList();
 7:
 8:     public void addItem(String id, String name, String price,
 9:         String quant) {
10:
11:         Item it = new Item(id, name, price, quant);
12:         catalog.add(it);
13:     }
14:
15:     public Item getItem(int i) {
16:         return (Item) catalog.get(i);
17:     }
18:
19:     public int getSize() {
20:         return catalog.size();
21:     }
22:
23:     public void sort() {
24:         Collections.sort(catalog);
25:     }
26: }
```

```
 1: package org.cadenhead.ecommerce;
 2:
 3: public class GiftShop {
 4:     public static void main(String[] arguments) {
 5:         Storefront store = new Storefront();
 6:         store.addItem("C01", "MUG", "9.99", "150");
 7:         store.addItem("C02", "LG MUG", "12.99", "82");
 8:         store.addItem("C03", "MOUSEPAD", "10.49", "800");
 9:         store.addItem("D01", "T SHIRT", "16.99", "90");
10:         store.sort();
11:
12:         for (int i = 0; i < store.getSize(); i++) {
13:             Item show = (Item) store.getItem(i);
14:             System.out.println("\nItem ID: " + show.getId() +
15:                 "\nName: " + show.getName() +
16:                 "\nRetail Price: $" + show.getRetail() +
17:                 "\nPrice: $" + show.getPrice() +
18:                 "\nQuantity: " + show.getQuantity());
19:         }
20:     }
21: }
```

```
package org.cadenhead.bureau;

public class Information {
    public int duration = 12;
    protected float rate = 3.15F;
    float average = 0.5F;
}
```

```
package org.cadenhead.bureau;

public class MoreInformation extends Information {
    public int quantity = 8;
}
```

```
package org.cadenhead.bureau.us;

import org.cadenhead.bureau.*;

public class EvenMoreInformation extends MoreInformation {
    public int quantity = 9;

    EvenMoreInformation() {
        super();
        int i1 = duration;
        float i2 = rate;
        float i3 = average;
    }
}
```

```
int status = loadTextFile();
if (status != 1) {
    // something unusual happened; report it
    switch (status) {
        case 2:
            System.out.println("File not found");
            break;
        case 3:
            System.out.println("Disk error");
            break;
        case 4:
            System.out.println("File corrupted");
            break;
        default:
            System.out.println("Error");
    }
} else {
    // file loaded OK; continue with program
}
```

```
Exception java.lang.InterruptedException
must be caught or it must be declared in the throws clause
of this method.
```

```
public SquareTool(String input) {
    try {
        float in = Float.parseFloat(input);
        // rest of method
    } catch (NumberFormatException nfe) {
        System.out.println(input + " is not a valid number.");
    }
}
```

```
try {
    float in = Float.parseFloat(input);
} catch (NumberFormatException nfe) {
    System.out.println("Oops: " + nfe.getMessage());
}
```

```
try {
    // code that might generate exceptions
} catch (IOException ioe) {
    System.out.println("Input/output error");
    System.out.println(ioe.getMessage());
} catch (ClassNotFoundException cnfe) {
    System.out.println("Class not found");
    System.out.println(cnfe.getMessage());
} catch (InterruptedException ie) {
    System.out.println("Program interrupted");
    System.out.println(ie.getMessage());
}
```

```
try {
    // code that reads a file from disk
} catch (EOFException | FileNotFoundException exc) {
    System.out.println("File error: " + exc.getMessage());
}
```

```
try {
    // code that reads a file from disk
} catch (IOException | EOFException | FileNotFoundException exc) {
    System.out.println("File error: " + exc.getMessage());
}
```

```
try {
    // code that reads a file from disk
} catch (EOFException | FileNotFoundException exc) {
    System.out.println("File error: " + exc.getMessage());
} catch (IOException ioe) {
    System.out.println("IO error: " + ioe.getMessage());
}
```

```
try {
    // code that reads a file from disk
} catch (EOFException | FileNotFoundException | IOException exc) {
    System.out.println("File error: " + exc.getMessage());
}
```

```
 1: package com.java21days;
 2:
 3: class HexReader {
 4:     String[] input = { "000A110D1D260219 ",
 5:         "78700F1318141E0C ",
 6:         "6A197D45B0FFFFFF " };
 7:
 8:     public static void main(String[] arguments) {
 9:         HexReader hex = new HexReader();
10:         for (int i = 0; i < hex.input.length; i++)
11:             hex.readLine(hex.input[i]);
12:     }
13:
14:     void readLine(String code) {
15:         try {
16:             for (int j = 0; j + 1 < code.length(); j += 2) {
17:                 String sub = code.substring(j, j + 2);
18:                 int num = Integer.parseInt(sub, 16);
19:                 if (num == 255) {
20:                     return;
21:                 }
22:                 System.out.print(num + " ");
23:             }
24:         } finally {
25:             System.out.println("**");
26:         }
27:         return;
28:     }
29: }
```

```
Socket digit = new Socket(host, 79);
BufferedReader in = new BufferedReader(
    new InputStreamReader(digit.getInputStream()));
```

```
try (Socket digit = new Socket(host, 79);
     BufferedReader in = new BufferedReader(
         new InputStreamReader(digit.getInputStream()));
     ) {

     // code goes here
} catch (IOException e) {
     System.out.println("IO Error:" + e.getMessage());
}
```

```
public void getPoint(int x, int y) throws NumberFormatException {
    // body of method
}
```

```
public void storePoint(int x, int y)
    throws NumberFormatException, EOFException {
        // body of method
}
```

```
public void loadPoint() throws IOException {
    // body of method
}
```

```
public WebRetriever() throws MalformedURLException {
    // body of constructor
}
```

```
public void readFloat(String input) throws NumberFormatException {
    float in = Float.parseFloat(input);
}
```

```
public class RadioPlayer {
    public void startPlaying() throws SoundException {
        // body of method
    }
}

public class StereoPlayer extends RadioPlayer {
    public void startPlaying() {
        // body of method
    }
}
```

```
void readFields() throws IOException {
    // body of method
}
```

```
void readFiles() throws SQLException {
    // body of method
}
```

```
NotInServiceException nise = new NotInServiceException();
throw nise;
```

```
NotInServiceException nise = new
    NotInServiceException("Database Not in Service");
throw nise;
```

```
public class SunSpotException extends Exception {
    public SunSpotException() {}

    public SunSpotException(String message) {
        super(message);
    }
}
```

```
public void readMessage() throws IOException {
    MessageReader mr = new MessageReader();

    try {
        mr.loadHeader();
    } catch (IOException e) {
        // do something to handle the
        // IO exception and then rethrow
        // the exception ...
        throw e;
    }
}
```

```
try {
    Thread.sleep(3000);
} catch (InterruptedException ie) {
    // do nothing
}
```

```
public class StockTicker implements Runnable {
    public void run() {
        // ...
    }
}
```

```
StockTicker tix = new StockTicker();
Thread tickerThread = new Thread(tix);
```

```
 1: package com.java21days;
 2:
 3: public class PrimeFinder implements Runnable {
 4:     public long target;
 5:     public long prime;
 6:     public boolean finished = false;
 7:     private Thread runner;
 8:
 9:     PrimeFinder(long inTarget) {
10:         target = inTarget;
11:         if (runner == null) {
12:             runner = new Thread(this);
13:             runner.start();
14:         }
15:     }
16:
17:     public void run() {
18:         long numPrimes = 0;
19:         long candidate = 2;
20:         while (numPrimes < target) {
```

```
21:             if (isPrime(candidate)) {
22:                 numPrimes++;
23:                 prime = candidate;
24:             }
25:             candidate++;
26:         }
27:         finished = true;
28:     }
29:
30:     boolean isPrime(long checkNumber) {
31:         double root = Math.sqrt(checkNumber);
32:         for (int i = 2; i <= root; i++) {
33:             if (checkNumber % i == 0)
34:                 return false;
35:         }
36:         return true;
37:     }
38: }
```

```
 1: package com.java21days;
 2:
 3: public class PrimeThreads {
 4:     public static void main(String[] arguments) {
 5:         PrimeThreads pt = new PrimeThreads(arguments);
 6:     }
 7:
 8:     public PrimeThreads(String[] arguments) {
 9:         PrimeFinder[] finder = new PrimeFinder[arguments.length];
10:         for (int i = 0; i < arguments.length; i++) {
11:             try {
12:                 long count = Long.parseLong(arguments[i]);
13:                 finder[i] = new PrimeFinder(count);
14:                 System.out.println("Looking for prime " + count);
15:             } catch (NumberFormatException nfe) {
16:                 System.out.println("Error: " + nfe.getMessage());
17:             }
18:         }
19:         boolean complete = false;
20:         while (!complete) {
21:             complete = true;
22:             for (int j = 0; j < finder.length; j++) {
```

```
23:                     if (finder[j] == null) continue;
24:                     if (!finder[j].finished) {
25:                         complete = false;
26:                     } else {
27:                         displayResult(finder[j]);
28:                         finder[j] = null;
29:                     }
30:                 }
31:                 try {
32:                     Thread.sleep(1000);
33:                 } catch (InterruptedException ie) {
34:                     // do nothing
35:                 }
36:             }
37:         }
38:
39:         private void displayResult(PrimeFinder finder) {
40:             System.out.println("Prime " + finder.target
41:                 + " is " + finder.prime);
42:         }
43: }
```

```
public void run() {
    Thread thisThread = Thread.currentThread();
    while (runner == thisThread) {
        // body of loop
    }
}
```

```
public class AverageValue {
    public static void main(String[] arguments) {
        float[] temps = new float[10];
        float sum = 0;
        int count = 0;
        int i;
        for (i = 0; i < arguments.length & i < 10; i++) {
            try {
                temps[i] = Float.parseFloat(arguments[i]);
                count++;
            } catch (NumberFormatException nfe) {
                System.out.println("Invalid input: " + arguments[i]);
            }
            sum += temps[i];
        }
        System.out.println("Average: " + (sum / i));
    }
}
```

```
while (users.hasNext()) {
    String ob = (String) users.next();
    System.out.println(ob);
}
```

```
class ConnectionAttributes {
    public static final int READABLE = 0;
    public static final int WRITABLE = 1;
    public static final int STREAMABLE = 2;
    public static final int FLEXIBLE = 3;
}
```

```
connex.set(ConnectionAttributes.WRITABLE);
connex.set(ConnectionAttributes.STREAMABLE);
connex.set(ConnectionAttributes.FLEXIBLE);

connex.clear(ConnectionAttributes.WRITABLE);
```

```
boolean isWriteable = connex.get(ConnectionAttributes.WRITABLE);
```

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class HolidaySked {
 6:     BitSet sked;
 7:
 8:     public HolidaySked() {
 9:         sked = new BitSet(365);
10:         int[] holiday = { 1, 15, 50, 148, 185, 246,
11:             281, 316, 326, 359 };
12:         for (int i = 0; i < holiday.length; i++) {
13:             addHoliday(holiday[i]);
14:         }
15:     }
16:
17:     public void addHoliday(int dayToAdd) {
18:         sked.set(dayToAdd);
19:     }
20:
21:     public boolean isHoliday(int dayToCheck) {
```

```
22:         boolean result = sked.get(dayToCheck);
23:         return result;
24:     }
25:
26:     public static void main(String[] arguments) {
27:         HolidaySked cal = new HolidaySked();
28:         if (arguments.length > 0) {
29:             try {
30:                 int whichDay = Integer.parseInt(arguments[0]);
31:                 if (cal.isHoliday(whichDay)) {
32:                     System.out.println("Day number " + whichDay +
33:                         " is a holiday.");
34:                 } else {
35:                     System.out.println("Day number " + whichDay +
36:                         " is not a holiday.");
37:                 }
38:             } catch (NumberFormatException nfe) {
39:                 System.out.println("Error: " + nfe.getMessage());
40:             }
41:         }
42:     }
43: }
```

```
ArrayList golfer = new ArrayList();
```

```
ArrayList golfer = new ArrayList(30);
```

```
String s1 = (String) golfer.get(0);
String s2 = (String) golfer.get(2);
```

```
boolean isThere = golfer.contains("Kerr");
```

```
Iterator it = golfer.iterator();
```

```
for (Iterator i = golfer.iterator(); i.hasNext(); ) {
    String name = (String) i.next();
    System.out.println(name);
}
```

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class CodeKeeper {
 6:     ArrayList list;
 7:     String[] codes = { "alpha", "lambda", "gamma", "delta", "zeta" };
 8:
 9:     public CodeKeeper(String[] userCodes) {
10:         list = new ArrayList();
11:         // load built-in codes
12:         for (int i = 0; i < codes.length; i++) {
13:             addCode(codes[i]);
14:         }
15:         // load user codes
16:         for (int j = 0; j < userCodes.length; j++) {
17:             addCode(userCodes[j]);
18:         }
19:         // display all codes
20:         for (Iterator ite = list.iterator(); ite.hasNext(); ) {
21:             String output = (String) ite.next();
22:             System.out.println(output);
23:        }
24:     }
25:
26:     private void addCode(String code) {
27:         if (!list.contains(code)) {
28:             list.add(code);
29:         }
30:     }
31:
32:     public static void main(String[] arguments) {
33:         CodeKeeper keeper = new CodeKeeper(arguments);
34:     }
35: }
```

```
Rectangle r1 = new Rectangle(0, 0, 5, 5);
look.put("small", r1);
Rectangle r2 = new Rectangle(0, 0, 15, 15);
look.put("medium", r2);
Rectangle r3 = new Rectangle(0, 0, 25, 25);
look.put("large", r3);
```

```
Rectangle r = (Rectangle) look.get("medium");
```

```
boolean isEmpty = look.isEmpty();
```

```
HashMap hash = new HashMap(20);
```

```
HashMap hash = new HashMap(20, 0.5F);
```

```
Rectangle box = new Rectangle(0, 0, 5, 5);
boolean isThere = hash.containsValue(box);
```

```
boolean isThere = hash.containsKey("Small");
```

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class ComicBooks {
 6:
 7:     public ComicBooks() {
 8:     }
 9:
10:     public static void main(String[] arguments) {
11:         // set up hash map
12:         HashMap quality = new HashMap();
13:         float price1 = 3.00F;
14:         quality.put("mint", price1);
15:         float price2 = 2.00F;
16:         quality.put("near mint", price2);
17:         float price3 = 1.50F;
18:         quality.put("very fine", price3);
19:         float price4 = 1.00F;
20:         quality.put("fine", price4);
21:         float price5 = 0.50F;
22:         quality.put("good", price5);
23:         float price6 = 0.25F;
24:         quality.put("poor", price6);
25:         // set up collection
26:         Comic[] comix = new Comic[3];
27:         comix[0] = new Comic("Amazing Spider-Man", "1A", "very fine",
28:             12_000.00F);
29:         comix[0].setPrice( (Float) quality.get(comix[0].condition) );
```

```
30:          comix[1] = new Comic("Incredible Hulk", "181", "near mint",
31:              680.00F);
32:          comix[1].setPrice( (Float) quality.get(comix[1].condition) );
33:          comix[2] = new Comic("Cerebus", "1A", "good", 190.00F);
34:          comix[2].setPrice( (Float) quality.get(comix[2].condition) );
35:          for (int i = 0; i < comix.length; i++) {
36:              System.out.println("Title: " + comix[i].title);
37:              System.out.println("Issue: " + comix[i].issueNumber);
38:              System.out.println("Condition: " + comix[i].condition);
39:              System.out.println("Price: $" + comix[i].price + "\n");
40:          }
41:      }
42: }
43:
44: class Comic {
45:      String title;
46:      String issueNumber;
47:      String condition;
48:      float basePrice;
49:      float price;
50:
51:      Comic(String inTitle, String inIssueNumber, String inCondition,
52:          float inBasePrice) {
53:
54:          title = inTitle;
55:          issueNumber = inIssueNumber;
56:          condition = inCondition;
57:          basePrice = inBasePrice;
58:      }
59:
60:      void setPrice(float factor) {
61:          price = basePrice * factor;
62:      }
63: }
```

```
comix[0].setPrice( (Float) quality.get(comix[0].condition) );
```

```
quality.put("near mint", 1.50F);
```

```
quality.put("near mint", "1.50");
```

```
comix[1].setPrice( (Float) quality.get(comix[1].condition) );
```

```
ArrayList<Integer> zipCodes = new ArrayList<>();
```

```
HashMap<String, Float> quality = new HashMap<String, Float>();
```

```
HashMap<String, Float> quality = new HashMap<>();
```

```
comix[1].setPrice(quality.get(comix[1].condition));
```

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class CodeKeeper2 {
 6:     ArrayList<String> list;
 7:     String[] codes = { "alpha", "lambda", "gamma", "delta", "zeta" };
 8:
 9:     public CodeKeeper2(String[] userCodes) {
10:         list = new ArrayList<>();
11:         // load built-in codes
12:         for (int i = 0; i < codes.length; i++) {
13:             addCode(codes[i]);
14:         }
15:         // load user codes
16:         for (int j = 0; j < userCodes.length; j++) {
17:             addCode(userCodes[j]);
18:         }
19:         // display all codes
20:         for (String code : list) {
21:             System.out.println(code);
22:         }
23:     }
24:
25:     private void addCode(String code) {
26:         if (!list.contains(code)) {
27:             list.add(code);
28:         }
29:     }
30:
31:     public static void main(String[] arguments) {
32:         CodeKeeper2 keeper = new CodeKeeper2(arguments);
33:     }
34: }
```

```
class ConnectionAttributes {
    public static final int READABLE = 0;
    public static final int WRITABLE = 1;
    public static final int STREAMABLE = 2;
    public static final int FLEXIBLE = 3;
}
```

```
setConnectionType(1);

setConnectionType(ConnectionAttributes.WRITABLE);
```

```
public class DirectionSetter {
    Compass current;
    public void setDirection(Compass dir) {
        current = dir;
    }

    public static void main(String[] arguments) {
        DirectionSetter app = new DirectionSetter();
        app.setDirection(Compass.WEST);
        System.out.println(app.current);
    }
}
```

```
public class Recursion {
    public int dex = -1;

    public Recursion() {
        dex = getValue(17);
    }

    public int getValue(int dexValue) {
        if (dexValue > 100) {
            return dexValue;
        } else {
            return getValue(dexValue * 2);
        }
    }

    public static void main(String[] arguments) {
        Recursion r = new Recursion();
        System.out.println(r.dex);
    }
}
```

```
public class FeedReader extends JFrame {
    // body of class
}
```

```
public class Payroll extends javax.swing.JFrame {
    public Payroll() {
        super("Edit Payroll");
        setSize(300, 100);
        setVisible(true);
    }
}
```

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

```
UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
 );
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class SimpleFrame extends JFrame {
 6:     public SimpleFrame() {
 7:         super("Frame Title");
 8:         setSize(300, 100);
 9:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10:         setLookAndFeel();
11:         setVisible(true);
12:     }
13:
14:     private static void setLookAndFeel() {
15:         try {
16:             UIManager.setLookAndFeel(
17:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
18:             );
19:         } catch (Exception exc) {
20:             // ignore error
21:         }
22:     }
23:
24:     public static void main(String[] arguments) {
25:         setLookAndFeel();
26:         SimpleFrame sf = new SimpleFrame();
27:     }
28: }
```

```
JButton play = new JButton("Play");
JButton stop = new JButton("Stop");
JButton rewind = new JButton("Rewind");
```

```
JButton quit = new JButton("Quit");
JPanel panel = new JPanel();
panel.add(quit);
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class ButtonFrame extends JFrame {
 6:     JButton load = new JButton("Load");
 7:     JButton save = new JButton("Save");
 8:     JButton unsubscribe = new JButton("Unsubscribe");
 9:
10:     public ButtonFrame() {
11:         super("Button Frame");
12:         setSize(340, 170);
13:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:         JPanel pane = new JPanel();
15:         pane.add(load);
16:         pane.add(save);
17:         pane.add(unsubscribe);
18:         add(pane);
19:         setVisible(true);
```

```
20:     }
21:
22:     private static void setLookAndFeel() {
23:         try {
24:             UIManager.setLookAndFeel(
25:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
26:             );
27:         } catch (Exception exc) {
28:             System.out.println(exc.getMessage());
29:         }
30:     }
31:
32:     public static void main(String[] arguments) {
33:         setLookAndFeel();
34:         ButtonFrame bf = new ButtonFrame();
35:     }
36: }
```

```
ImageIcon subscribe = new ImageIcon("subscribe.gif");
JButton button = new JButton(subscribe);
JPanel pane = new JPanel();
pane.add(button);
add(pane);
setVisible(true);
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class IconFrame extends JFrame {
 6:     JButton load, save, subscribe, unsubscribe;
 7:
 8:     public IconFrame() {
 9:         super("Icon Frame");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         JPanel panel = new JPanel();
12:         // create icons
13:         ImageIcon loadIcon = new ImageIcon("load.gif");
14:         ImageIcon saveIcon = new ImageIcon("save.gif");
15:         ImageIcon subscribeIcon = new ImageIcon("subscribe.gif");
16:         ImageIcon unsubscribeIcon = new ImageIcon("unsubscribe.gif");
17:         // create buttons
18:         load = new JButton("Load", loadIcon);
19:         save = new JButton("Save", saveIcon);
```

```
20:         subscribe = new JButton("Subscribe", subscribeIcon);
21:         unsubscribe = new JButton("Unsubscribe", unsubscribeIcon);
22:         // add buttons to panel
23:         panel.add(load);
24:         panel.add(save);
25:         panel.add(subscribe);
26:         panel.add(unsubscribe);
27:         // add the panel to a frame
28:         add(panel);
29:         pack();
30:         setVisible(true);
31:     }
32:
33:     public static void main(String[] arguments) {
34:         IconFrame ike = new IconFrame();
35:     }
36: }
```

```
JLabel feedsLabel = new JLabel("Feeds: ", SwingConstants.LEFT);
JLabel urlLabel = new JLabel("URL: ", SwingConstants.CENTER);
JLabel dateLabel = new JLabel("Date: ", SwingConstants.RIGHT);
```

```
JTextField rssUrl = new JTextField(60);
JTextField rssUrl2 = new JTextField("Enter feed URL here", 60);
```

```
JPasswordField codePhrase = new JPasswordField(20);
codePhrase.setEchoChar('#');
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class Authenticator extends javax.swing.JFrame {
 6:     JTextField username = new JTextField(15);
 7:     JPasswordField password = new JPasswordField(15);
 8:     JTextArea comments = new JTextArea(4, 15);
 9:     JButton ok = new JButton("OK");
10:     JButton cancel = new JButton("Cancel");
11:
12:     public Authenticator() {
13:         super("Account Information");
14:         setSize(300, 220);
15:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16:
17:         JPanel pane = new JPanel();
18:         JLabel usernameLabel = new JLabel("Username: ");
19:         JLabel passwordLabel = new JLabel("Password: ");
20:         JLabel commentsLabel = new JLabel("Comments: ");
21:         comments.setLineWrap(true);
22:         comments.setWrapStyleWord(true);
23:         pane.add(usernameLabel);
24:         pane.add(username);
25:         pane.add(passwordLabel);
```

```
26:         pane.add(password);
27:         pane.add(commentsLabel);
28:         pane.add(comments);
29:         pane.add(ok);
30:         pane.add(cancel);
31:         add(pane);
32:         setVisible(true);
33:     }
34:
35:     private static void setLookAndFeel() {
36:         try {
37:             UIManager.setLookAndFeel(
38:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
39:             );
40:         } catch (Exception exc) {
41:             System.out.println(exc.getMessage());
42:         }
43:     }
44:
45:     public static void main(String[] arguments) {
46:         Authenticator.setLookAndFeel();
47:         Authenticator auth = new Authenticator();
48:     }
49: }
```

```
JPanel pane = new JPanel();
JTextArea comments = new JTextArea(4, 15);
JScrollPane scroll = new JScrollPane(comments,
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
pane.add(scroll);
add(pane);
```

```
ButtonGroup choice = new ButtonGroup();
```

```
ButtonGroup saveFormat = new ButtonGroup();
JRadioButton s1 = new JRadioButton("JSON", false);
saveFormat.add(s1);
JRadioButton s2 = new JRadioButton("XML", true);
saveFormat.add(s2);
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class FormatFrame extends JFrame {
 6:     JRadioButton[] teams = new JRadioButton[4];
 7:
 8:     public FormatFrame() {
 9:         super("Choose an Output Format");
10:         setSize(320, 120);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         teams[0] = new JRadioButton("Atom");
13:         teams[1] = new JRadioButton("RSS 0.92");
14:         teams[2] = new JRadioButton("RSS 1.0");
15:         teams[3] = new JRadioButton("RSS 2.0", true);
16:         JPanel panel = new JPanel();
17:         JLabel chooseLabel = new JLabel(
18:             "Choose an output format for syndicated news items.");
19:         panel.add(chooseLabel);
```

```
20:         ButtonGroup group = new ButtonGroup();
21:         for (JRadioButton team : teams) {
22:             group.add(team);
23:             panel.add(team);
24:         }
25:         add(panel);
26:         setVisible(true);
27:     }
28:
29:     private static void setLookAndFeel() {
30:         try {
31:             UIManager.setLookAndFeel(
32:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
33:             );
34:         } catch (Exception exc) {
35:             System.out.println(exc.getMessage());
36:         }
37:     }
38:
39:     public static void main(String[] arguments) {
40:         FormatFrame.setLookAndFeel();
41:         FormatFrame ff = new FormatFrame();
42:     }
43: }
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class FormatFrame2 extends JFrame {
 6:     String[] formats = { "Atom", "RSS 0.92", "RSS 1.0", "RSS 2.0" };
 7:     JComboBox formatBox = new JComboBox(formats);
 8:
 9:     public FormatFrame2() {
10:         super("Choose a Format");
11:         setSize(220, 150);
12:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13:         JPanel pane = new JPanel();
14:         JLabel formatLabel = new JLabel("Output formats:");
15:         pane.add(formatLabel);
16:         pane.add(formatBox);
17:         add(pane);
18:         setVisible(true);
19:     }
20:
21:     private static void setLookAndFeel() {
22:         try {
23:             UIManager.setLookAndFeel(
24:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
25:             );
26:         } catch (Exception exc) {
27:             System.out.println(exc.getMessage());
28:         }
29:     }
30:
31:     public static void main(String[] arguments) {
32:         FormatFrame2.setLookAndFeel();
33:         FormatFrame2 ff = new FormatFrame2();
34:     }
35: }
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class Subscriptions extends JFrame {
 6:     String[] subs = { "Burningbird", "Freeform Goodness",
 7:         "Ideoplex", "Inessential", "Intertwingly", "Now This",
 8:         "Rasterweb", "RC3", "Whole Lotta Nothing", "Workbench" };
 9:     JList<String> subList = new JList<>(subs);
10:
11:     public Subscriptions() {
12:         super("Subscriptions");
13:         setSize(150, 335);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         JPanel panel = new JPanel();
16:         JLabel subLabel = new JLabel("RSS Subscriptions:");
17:         panel.add(subLabel);
18:         subList.setVisibleRowCount(8);
19:         JScrollPane scroller = new JScrollPane(subList);
20:         panel.add(scroller);
```

```
21:         add(panel);
22:         setVisible(true);
23:     }
24:
25:     private static void setLookAndFeel() {
26:         try {
27:             UIManager.setLookAndFeel(
28:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
29:             );
30:         } catch (Exception exc) {
31:             System.out.println(exc.getMessage());
32:         }
33:     }
34:
35:     public static void main(String[] arguments) {
36:         Subscriptions.setLookAndFeel();
37:         Subscriptions app = new Subscriptions();
38:     }
39: }
```

```
double d100 = Math.random() * 100;
```

```
import javax.swing.*;

public class Display extends JFrame {
    public Display() {
        super("Display");
        // answer goes here
        JLabel hello = new JLabel("Hello");
        JPanel pane = new JPanel();
        add(hello);
        pack();
        setVisible(true);
    }

    public static void main(String[] arguments) {
        Display ds = new Display();
    }
}
```

```
int response = JOptionPane.showConfirmDialog(null,
    "Should I delete all of your irreplaceable personal files?");
```

```
int response = JOptionPane.showConfirmDialog(null,
    "Error reading file. Want to try again?",
    "File Input Error",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.ERROR_MESSAGE);
```

```
String response = JOptionPane.showInputDialog(null,
    "Enter your name:");
```

```
String response = JOptionPane.showInputDialog(null,
     "What is your ZIP code?",
     "Enter ZIP Code",
     JOptionPane.QUESTION_MESSAGE);
```

```
JOptionPane.showMessageDialog(null,
    "The program has been uninstalled.");
```

```
JOptionPane.showMessageDialog(null,
    "An asteroid has destroyed the Earth.",
    "Asteroid Destruction Alert",
   JOptionPane.WARNING_MESSAGE);
```

```
String[] gender = {
     "Male",
     "Female",
     "None of Your Business"
};
int response = JOptionPane.showOptionDialog(null,
     "What is your gender?",
     "Gender",
     0,
     JOptionPane.INFORMATION_MESSAGE,
     null,
     gender,
     gender[2]);
System.out.println("You chose " + gender[response]);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.GridLayout;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class FeedInfo extends JFrame {
 8:     private JLabel nameLabel = new JLabel("Name: ",
 9:         SwingConstants.RIGHT);
10:     private JTextField name;
11:     private JLabel urlLabel = new JLabel("URL: ",
12:         SwingConstants.RIGHT);
13:     private JTextField url;
14:     private JLabel typeLabel = new JLabel("Type: ",
15:         SwingConstants.RIGHT);
16:     private JTextField type;
17:
18:     public FeedInfo() {
19:         super("Feed Information");
20:         setSize(400, 145);
21:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22:         setLookAndFeel();
23:         // Site name
24:         String response1 = JOptionPane.showInputDialog(null,
25:             "Enter the site name:");
26:         name = new JTextField(response1, 20);
27:
28:         // Site address
29:         String response2 = JOptionPane.showInputDialog(null,
30:             "Enter the site address:");
31:         url = new JTextField(response2, 20);
32:
33:         // Site type
34:         String[] choices = { "Personal", "Commercial", "Unknown" };
35:         int response3 = JOptionPane.showOptionDialog(null,
```

```
36:             "What type of site is it?",
37:             "Site Type",
38:             0,
39:             JOptionPane.QUESTION_MESSAGE,
40:             null,
41:             choices,
42:             choices[0]);
43:         type = new JTextField(choices[response3], 20);
44:
45:         setLayout(new GridLayout(3, 2));
46:         add(nameLabel);
47:         add(name);
48:         add(urlLabel);
49:         add(url);
50:         add(typeLabel);
51:         add(type);
52:         setLookAndFeel();
53:         setVisible(true);
54:     }
55:
```

```
56:     private void setLookAndFeel() {
57:         try {
58:             UIManager.setLookAndFeel(
59:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
60:             );
61:             SwingUtilities.updateComponentTreeUI(this);
62:         } catch (Exception e) {
63:             System.err.println("Couldn't use the system "
64:                 + "look and feel: " + e);
65:         }
66:     }
67:
68:     public static void main(String[] arguments) {
69:         FeedInfo frame = new FeedInfo();
70:     }
71: }
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class Slider extends JFrame {
 6:
 7:     public Slider() {
 8:         super("Slider");
 9:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10:         setLookAndFeel();
11:         JSlider pick = new JSlider(JSlider.HORIZONTAL, 0, 30, 5);
12:         pick.setMajorTickSpacing(10);
13:         pick.setMinorTickSpacing(1);
14:         pick.setPaintTicks(true);
15:         pick.setPaintLabels(true);
16:         add(pick);
17:         pack();
18:         setVisible(true);
19:     }
20:
```

```
21:     private void setLookAndFeel() {
22:         try {
23:             UIManager.setLookAndFeel(
24:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
25:             );
26:             SwingUtilities.updateComponentTreeUI(this);
27:         } catch (Exception e) {
28:             System.err.println("Couldn't use the system "
29:                 + "look and feel: " + e);
30:         }
31:     }
32:
33:     public static void main(String[] arguments) {
34:         Slider frame = new Slider();
35:     }
36: }
```

```
JTextArea textBox = new JTextArea(7, 30);
JScrollPane scroller = new JScrollPane(textBox);
mainPane.add(scroller);
```

```
Dimension pref = new Dimension(350, 100);
scroller.setPreferredSize(pref);
```

```
JScrollPane scroller = new JScrollPane(textBox,
    VERTICAL_SCROLLBAR_ALWAYS,
    HORIZONTAL_SCROLLBAR_NEVER);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class FeedBar extends JFrame {
 7:
 8:     public FeedBar() {
 9:         super("FeedBar");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         // create icons
13:         ImageIcon loadIcon = new ImageIcon("load.gif");
14:         ImageIcon saveIcon = new ImageIcon("save.gif");
15:         ImageIcon subIcon = new ImageIcon("subscribe.gif");
16:         ImageIcon unsubIcon = new ImageIcon("unsubscribe.gif");
17:         // create buttons
18:         JButton load = new JButton("Load", loadIcon);
19:         JButton save = new JButton("Save", saveIcon);
20:         JButton sub = new JButton("Subscribe", subIcon);
21:         JButton unsub = new JButton("Unsubscribe", unsubIcon);
22:         // add buttons to toolbar
23:         JToolBar bar = new JToolBar();
24:         bar.add(load);
25:         bar.add(save);
26:         bar.add(sub);
```

```
27:         bar.add(unsub);
28:         // prepare user interface
29:         JTextArea edit = new JTextArea(8, 40);
30:         JScrollPane scroll = new JScrollPane(edit);
31:         BorderLayout bord = new BorderLayout();
32:         setLayout(bord);
33:         add("North", bar);
34:         add("Center", scroll);
35:         pack();
37:         setVisible(true);
38:     }
39:
40:     private void setLookAndFeel() {
41:         try {
42:             UIManager.setLookAndFeel(
43:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
44:             );
45:             SwingUtilities.updateComponentTreeUI(this);
46:         } catch (Exception e) {
47:             System.err.println("Couldn't use the system "
48:                 + "look and feel: " + e);
49:         }
50:     }
51:
52:     public static void main(String[] arguments) {
53:         FeedBar frame = new FeedBar();
54:     }
55: }
```

```
int filesDone = getNumberOfFiles();
install.setValue(filesDone);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class ProgressMonitor extends JFrame {
 7:
 8:     JProgressBar current;
 9:     JTextArea out;
10:     JButton find;
11:     int num = 0;
12:
13:     public ProgressMonitor() {
14:         super("Progress Monitor");
15:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16:         setLookAndFeel();
17:         setSize(205, 68);
18:         setLayout(new FlowLayout());
19:         current = new JProgressBar(0, 2000);
20:         current.setValue(0);
21:         current.setStringPainted(true);
22:         add(current);
23:     }
24:
25:     public void iterate() {
26:         while (num < 2000) {
27:             current.setValue(num);
```

```
28:                 try {
29:                     Thread.sleep(1000);
30:                 } catch (InterruptedException e) { }
31:                 num += 95;
32:             }
33:         }
34:
35:         private void setLookAndFeel() {
36:             try {
37:                 UIManager.setLookAndFeel(
38:                     "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
39:                 );
40:                 SwingUtilities.updateComponentTreeUI(this);
41:             } catch (Exception e) {
42:                 System.err.println("Couldn't use the system "
43:                     + "look and feel: " + e);
44:             }
45:         }
46:
47:         public static void main(String[] arguments) {
48:             ProgressMonitor frame = new ProgressMonitor();
49:             frame.setVisible(true);
50:             frame.iterate();
51:         }
52: }
```

```
JMenuItem j1 = new JMenuItem("Open");
JMenuItem j2 = new JMenuItem("Save");
JMenuItem j3 = new JMenuItem("Save as Template");
JMenuItem j4 = new JMenuItem("Page Setup");
JMenuItem j5 = new JMenuItem("Print");
JMenuItem j6 = new JMenuItem("Use as Default Message Style");
JMenuItem j7 = new JMenuItem("Close");
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class FeedBar2 extends JFrame {
 7:
 8:     public FeedBar2() {
 9:         super("FeedBar 2");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         // create icons
13:         ImageIcon loadIcon = new ImageIcon("load.gif");
14:         ImageIcon saveIcon = new ImageIcon("save.gif");
15:         ImageIcon subIcon = new ImageIcon("subscribe.gif");
16:         ImageIcon unsubIcon = new ImageIcon("unsubscribe.gif");
17:         // create buttons
18:         JButton load = new JButton("Load", loadIcon);
19:         JButton save = new JButton("Save", saveIcon);
20:         JButton sub = new JButton("Subscribe", subIcon);
21:         JButton unsub = new JButton("Unsubscribe", unsubIcon);
22:         // add buttons to toolbar
23:         JToolBar bar = new JToolBar();
24:         bar.add(load);
25:         bar.add(save);
26:         bar.add(sub);
27:         bar.add(unsub);
28:         // create menu
29:         JMenuItem j1 = new JMenuItem("Load");
30:         JMenuItem j2 = new JMenuItem("Save");
31:         JMenuItem j3 = new JMenuItem("Subscribe");
32:         JMenuItem j4 = new JMenuItem("Unsubscribe");
```

```
33:         JMenuBar menubar = new JMenuBar();
34:         JMenu menu = new JMenu("Feeds");
35:         menu.add(j1);
36:         menu.add(j2);
37:         menu.addSeparator();
38:         menu.add(j3);
39:         menu.add(j4);
40:         menubar.add(menu);
41:         // prepare user interface
42:         JTextArea edit = new JTextArea(8, 40);
43:         JScrollPane scroll = new JScrollPane(edit);
44:         BorderLayout bord = new BorderLayout();
45:         setLayout(bord);
46:         add("North", bar);
47:         add("Center", scroll);
48:         setJMenuBar(menubar);
49:         pack();
50:         setVisible(true);
51:     }
52:
```

```
53:     private void setLookAndFeel() {
54:         try {
55:             UIManager.setLookAndFeel(
56:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
57:             );
58:             SwingUtilities.updateComponentTreeUI(this);
59:         } catch (Exception e) {
60:             System.err.println("Couldn't use the system "
61:                 + "look and feel: " + e);
62:         }
63:     }
64:
65:     public static void main(String[] arguments) {
66:         FeedBar2 frame = new FeedBar2();
67:     }
68: }
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class TabPanels extends JFrame {
 7:
 8:     public TabPanels() {
 9:         super("Tabbed Panes");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         setSize(480, 218);
13:         JPanel mainSettings = new JPanel();
14:         JPanel advancedSettings = new JPanel();
15:         JPanel privacySettings = new JPanel();
16:         JPanel emailSettings = new JPanel();
17:         JPanel securitySettings = new JPanel();
18:         JTabbedPane tabs = new JTabbedPane();
19:         tabs.addTab("Main", mainSettings);
20:         tabs.addTab("Advanced", advancedSettings);
21:         tabs.addTab("Privacy", privacySettings);
22:         tabs.addTab("E-mail", emailSettings);
23:         tabs.addTab("Security", securitySettings);
```

```
24:          add(tabs);
25:          setVisible(true);
26:      }
27:
28:      private void setLookAndFeel() {
29:          try {
30:              UIManager.setLookAndFeel(
31:                  "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
32:              );
33:              SwingUtilities.updateComponentTreeUI(this);
34:          } catch (Exception e) {
35:              System.err.println("Couldn't use the system "
36:                  + "look and feel: " + e);
37:          }
38:      }
39:
40:      public static void main(String[] arguments) {
41:          TabPanels frame = new TabPanels();
42:      }
43: }
```

```
import java.awt.*;
import javax.swing.*;

public class AskFrame extends JFrame {
    public AskFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JSlider value = new JSlider(0, 255, 100);
        add(value);
        setSize(450, 150);
        setVisible(true);
        super();
    }

    public static void main(String[] arguments) {
        AskFrame af = new AskFrame();
    }
}
```

```
FlowLayout flo = new FlowLayout();
```

```
import java.awt.*;
import javax.swing.*;

public class Starter extends JFrame {

    public Starter() {
        super("Example Frame");
        FlowLayout manager = new FlowLayout();
        setLayout(manager);
        // add components here
    }
}
```

```
FlowLayout righty = new FlowLayout(FlowLayout.RIGHT);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class Alphabet extends JFrame {
 8:
 9:     public Alphabet() {
10:         super("Alphabet");
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         setLookAndFeel();
13:         setSize(360, 120);
14:         FlowLayout lm = new FlowLayout(FlowLayout.LEFT);
15:         setLayout(lm);
16:         JButton a = new JButton("Alibi");
17:         JButton b = new JButton("Burglar");
18:         JButton c = new JButton("Corpse");
19:         JButton d = new JButton("Deadbeat");
20:         JButton e = new JButton("Evidence");
21:         JButton f = new JButton("Fugitive");
22:         add(a);
23:         add(b);
24:         add(c);
```

```
25:         add(d);
26:         add(e);
27:         add(f);
28:         setVisible(true);
29:     }
30:
31:     private void setLookAndFeel() {
32:         try {
33:             UIManager.setLookAndFeel(
34:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
35:             );
36:             SwingUtilities.updateComponentTreeUI(this);
37:         } catch (Exception exc) {
38:             System.err.println("Couldn't use the system "
39:                 + "look and feel: " + exc);
40:         }
41:     }
42:
43:     public static void main(String[] arguments) {
44:         Alphabet frame = new Alphabet();
45:     }
46: }
```

```
FlowLayout flo = new FlowLayout(FlowLayout.CENTER, 30, 10);
```

```
JPanel optionPane = new JPanel();
BoxLayout box = new BoxLayout(optionPane, BoxLayout.Y_AXIS);
optionPane.setLayout(box);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class Stacker extends JFrame {
 7:     public Stacker() {
 8:         super("Stacker");
 9:         setSize(430, 150);
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         // create top panel
13:         JPanel commandPane = new JPanel();
14:         BoxLayout horizontal = new BoxLayout(commandPane,
15:             BoxLayout.X_AXIS);
16:         commandPane.setLayout(horizontal);
17:         JButton subscribe = new JButton("Subscribe");
18:         JButton unsubscribe = new JButton("Unsubscribe");
19:         JButton refresh = new JButton("Refresh");
20:         JButton save = new JButton("Save");
21:         commandPane.add(subscribe);
22:         commandPane.add(unsubscribe);
23:         commandPane.add(refresh);
24:         commandPane.add(save);
25:         // create bottom panel
```

```
26:         JPanel textPane = new JPanel();
27:         JTextArea text = new JTextArea(4, 70);
28:         JScrollPane scrollPane = new JScrollPane(text);
29:         // put them together
30:         FlowLayout flow = new FlowLayout();
31:         setLayout(flow);
32:         add(commandPane);
33:         add(scrollPane);
34:         setVisible(true);
35:     }
36:
37:     private void setLookAndFeel() {
38:         try {
39:             UIManager.setLookAndFeel(
40:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
41:             );
42:             SwingUtilities.updateComponentTreeUI(this);
43:         } catch (Exception exc) {
44:             System.err.println("Couldn't use the system "
45:                 + "look and feel: " + exc);
46:         }
47:     }
48:
49:     public static void main(String[] arguments) {
50:         Stacker st = new Stacker();
51:     }
52: }
```

```
GridLayout gr = new GridLayout(10, 3);
```

```
GridLayout gr2 = new GridLayout(10, 3, 5, 8);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class Bunch extends JFrame {
 8:
 9:     public Bunch() {
10:         super("Bunch");
11:         setSize(260, 260);
12:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13:         setLookAndFeel();
14:         JPanel pane = new JPanel();
15:         GridLayout family = new GridLayout(3, 3, 10, 10);
16:         pane.setLayout(family);
17:         JButton marcia = new JButton("Marcia");
18:         JButton carol = new JButton("Carol");
19:         JButton greg = new JButton("Greg");
20:         JButton jan = new JButton("Jan");
21:         JButton alice = new JButton("Alice");
22:         JButton peter = new JButton("Peter");
23:         JButton cindy = new JButton("Cindy");
24:         JButton mike = new JButton("Mike");
25:         JButton bobby = new JButton("Bobby");
26:         pane.add(marcia);
```

```
27:         pane.add(carol);
28:         pane.add(greg);
29:         pane.add(jan);
30:         pane.add(alice);
31:         pane.add(peter);
32:         pane.add(cindy);
33:         pane.add(mike);
34:         pane.add(bobby);
35:         add(pane);
36:         setVisible(true);
37:     }
38:
39:     private void setLookAndFeel() {
40:         try {
41:             UIManager.setLookAndFeel(
42:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
43:             );
44:             SwingUtilities.updateComponentTreeUI(this);
45:         } catch (Exception exc) {
46:             System.err.println("Couldn't use the system "
47:                 + "look and feel: " + exc);
48:         }
49:     }
50:
51:     public static void main(String[] arguments) {
52:         Bunch frame = new Bunch();
53:     }
54: }
```

```
JButton quitButton = new JButton("quit");
add(quitButton, BorderLayout.NORTH);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class Border extends JFrame {
 7:
 8:     public Border() {
 9:         super("Border");
10:         setSize(240, 280);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         setLookAndFeel();
13:         setLayout(new BorderLayout());
14:         JButton nButton = new JButton("North");
15:         JButton sButton = new JButton("South");
16:         JButton eButton = new JButton("East");
17:         JButton wButton = new JButton("West");
18:         JButton cButton = new JButton("Center");
19:         add(nButton, BorderLayout.NORTH);
20:         add(sButton, BorderLayout.SOUTH);
21:         add(eButton, BorderLayout.EAST);
22:         add(wButton, BorderLayout.WEST);
23:         add(cButton, BorderLayout.CENTER);
```

```
24:         setVisible(true);
25:     }
26:
27:     private void setLookAndFeel() {
28:         try {
29:             UIManager.setLookAndFeel(
30:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
31:             );
32:             SwingUtilities.updateComponentTreeUI(this);
33:         } catch (Exception exc) {
34:             System.err.println("Couldn't use the system "
35:                 + "look and feel: " + exc);
36:         }
37:     }
38:
39:     public static void main(String[] arguments) {
40:         Border frame = new Border();
41:     }
42: }
```

```
BorderLayout bl = new BorderLayout();
setLayout(bl);
```

```
FlowLayout flo = new FlowLayout();
pane.setLayout(flo);
```

```
JTextField nameField = new JTextField(80);
pane.add(nameField);
```

```
CardLayout cc = new CardLayout();
```

```
SurveyPanel[] ask = new SurveyPanel[3];
CardLayout cards = new CardLayout();
```

```
setLayout(cards);
String question1 = "What is your gender?";
String[] resp1 = { "female", "male", "not telling" };
ask[0] = new SurveyPanel(question1, resp1, 2);
add(ask[0], "Card 0");
```

```
SurveyPanel(String ques, String[] resp, int def) {
    question = new JLabel(ques);
    response = new JRadioButton[resp.length];
    // more to come
}
```

```
JPanel sub1 = new JPanel();
JLabel quesLabel = new JLabel(ques);
sub1.add(quesLabel);
```

```
JPanel sub2 = new JPanel();
for (int i = 0; i < resp.length; i++) {
    if (def == i) {
        response[i] = new JRadioButton(resp[i], true);
    } else {
        response[i] = new JRadioButton(resp[i], false);
    }
    group.add(response[i]);
    sub2.add(response[i]);
}
```

```
GridLayout grid = new GridLayout(3, 1);
setLayout(grid);
add(sub1);
add(sub2);
add(sub3);
```

```
void setFinalQuestion(boolean finalQuestion) {
    if (finalQuestion) {
        nextButton.setEnabled(false);
        finalButton.setEnabled(true);
    }
}
```

```
public class SurveyWizard extends JPanel implements ActionListener {
    // more to come
}
```

```
ask[0].nextButton.addActionListener(this);
ask[0].finalButton.addActionListener(this);
```

```
public void actionPerformed(Action evt) {
    // more to come
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class SurveyWizard extends JPanel implements ActionListener {
 8:     int currentCard = 0;
 9:     CardLayout cards = new CardLayout();
10:     SurveyPanel[] ask = new SurveyPanel[3];
11:
12:     public SurveyWizard() {
13:         super();
14:         setSize(240, 140);
15:         setLayout(cards);
16:         // set up survey
17:         String question1 = "What is your gender?";
18:         String[] resp1 = { "female", "male", "not telling" };
19:         ask[0] = new SurveyPanel(question1, resp1, 2);
20:         String question2 = "What is your age?";
21:         String[] resp2 = { "Under 25", "25-34", "35-54",
22:             "Over 54" };
23:         ask[1] = new SurveyPanel(question2, resp2, 1);
24:         String question3 = "How often do you exercise each week?";
25:         String[] resp3 = { "Never", "1-3 times", "More than 3" };
26:         ask[2] = new SurveyPanel(question3, resp3, 1);
27:         ask[2].setFinalQuestion(true);
28:         addListeners();
29:     }
30:
```

```
31:     private void addListeners() {
32:         for (int i = 0; i < ask.length; i++) {
33:             ask[i].nextButton.addActionListener(this);
34:             ask[i].finalButton.addActionListener(this);
35:             add(ask[i], "Card " + i);
36:         }
37:     }
38:
39:     public void actionPerformed(ActionEvent evt) {
40:         currentCard++;
41:         if (currentCard >= ask.length) {
42:             System.exit(0);
43:         }
44:         cards.show(this, "Card " + currentCard);
45:     }
46: }
47:
48: class SurveyPanel extends JPanel {
49:     JLabel question;
50:     JRadioButton[] response;
51:     JButton nextButton = new JButton("Next");
52:     JButton finalButton = new JButton("Finish");
53:
54:     SurveyPanel(String ques, String[] resp, int def) {
55:         super();
56:         setSize(160, 110);
57:         question = new JLabel(ques);
58:         response = new JRadioButton[resp.length];
59:         JPanel sub1 = new JPanel();
```

```
60:         ButtonGroup group = new ButtonGroup();
61:         JLabel quesLabel = new JLabel(ques);
62:         sub1.add(quesLabel);
63:         JPanel sub2 = new JPanel();
64:         for (int i = 0; i < resp.length; i++) {
65:             if (def == i) {
66:                 response[i] = new JRadioButton(resp[i], true);
67:             } else {
68:                 response[i] = new JRadioButton(resp[i], false);
69:             }
70:             group.add(response[i]);
71:             sub2.add(response[i]);
72:         }
73:         JPanel sub3 = new JPanel();
74:         nextButton.setEnabled(true);
75:         sub3.add(nextButton);
76:         finalButton.setEnabled(false);
77:         sub3.add(finalButton);
78:         GridLayout grid = new GridLayout(3, 1);
79:         setLayout(grid);
80:         add(sub1);
81:         add(sub2);
82:         add(sub3);
83:     }
84:
85:     void setFinalQuestion(boolean finalQuestion) {
86:         if (finalQuestion) {
87:             nextButton.setEnabled(false);
88:             finalButton.setEnabled(true);
89:         }
90:     }
91: }
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import javax.swing.*;
 5:
 6: public class SurveyFrame extends JFrame {
 7:     public SurveyFrame() {
 8:         super("Survey");
 9:         setSize(290, 140);
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setLookAndFeel();
12:         SurveyWizard wiz = new SurveyWizard();
13:         add(wiz);
14:         setVisible(true);
15:     }
16:
17:     private void setLookAndFeel() {
18:         try {
19:             UIManager.setLookAndFeel(
20:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
21:             );
22:             SwingUtilities.updateComponentTreeUI(this);
23:         } catch (Exception exc) {
24:             System.err.println("Couldn't use the system "
25:                 + "look and feel: " + exc);
26:         }
27:     }
28:
29:     public static void main(String[] arguments) {
30:         SurveyFrame surv = new SurveyFrame();
31:     }
32: }
```

```
Insets whitespace = new Insets(20, 13, 20, 13);
```

```
public Insets getInsets() {
    return new Insets(10, 30, 10, 30);
}
```

```
import java.awt.*;
import javax.swing.*;

public class Absolute extends JFrame {
    public Absolute() {
        super("Example");
        setSize(300, 300);
        setLayout(null);
        JButton myButton = new JButton("Click Me");
        myButton.setBounds(new Rectangle(10, 10, 120, 30));
        add(myButton);
        setVisible(true);
    }

    public static void main(String[] arguments) {
        Absolute ex = new Absolute();
    }
}
```

```
import java.awt.*;
import javax.swing.*;

public class ThreeButtons extends JFrame {
    public ThreeButtons() {
        super("Program");
        setSize(350, 225);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JButton alpha = new JButton("Alpha");
        JButton beta = new JButton("Beta");
        JButton gamma = new JButton("Gamma");
        // answer goes here
        add(alpha);
        add(beta);
        add(gamma);
        pack();
        setVisible(true);
    }

    public static void main(String[] arguments) {
        ThreeButtons b3 = new ThreeButtons();
    }
}
```

```
public class Suspense extends JFrame implements ActionListener,
     TextListener {
    // body of class
}
```

```
JButton zap = new JButton("Zap");
zap.addActionListener(this);
```

```
public void actionPerformed(ActionEvent event) {
    // handle event here
}
```

```
public void actionPerformed(ActionEvent event) {
    Object source = evt.getSource();
}
```

```
void actionPerformed(ActionEvent event) {
    Object source = event.getSource();
    if (source instanceof JTextField) {
        calculateScore();
    } else if (source instanceof JButton) {
        quit();
    }
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.event.*;
 4: import javax.swing.*;
 5: import java.awt.*;
 6:
 7: public class TitleBar extends JFrame implements ActionListener {
 8:     JButton b1;
 9:     JButton b2;
10:
11:     public TitleBar() {
12:         super("Title Bar");
13:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:         setLookAndFeel();
15:         b1 = new JButton("Rosencrantz");
16:         b2 = new JButton("Guildenstern");
17:         b1.addActionListener(this);
18:         b2.addActionListener(this);
19:         FlowLayout flow = new FlowLayout();
20:         setLayout(flow);
21:         add(b1);
22:         add(b2);
23:         pack();
24:         setVisible(true);
25:     }
26:
```

```
27:     public void actionPerformed(ActionEvent evt) {
28:         Object source = evt.getSource();
29:         if (source == b1) {
30:             setTitle("Rosencrantz");
31:         } else if (source == b2) {
32:             setTitle("Guildenstern");
33:         }
34:         repaint();
35:     }
36:
37:     private void setLookAndFeel() {
38:         try {
39:             UIManager.setLookAndFeel(
40:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
41:             );
42:             SwingUtilities.updateComponentTreeUI(this);
43:         } catch (Exception exc) {
44:             System.err.println("Couldn't use the system "
45:                 + "look and feel: " + exc);
46:         }
47:     }
48:
49:     public static void main(String[] arguments) {
50:         TitleBar frame = new TitleBar();
51:     }
52: }
```

```
public void actionPerformed(ActionEvent event) {
    // ...
}
```

```
JButton sort = new JButton("Sort");
JMenuItem menuSort = new JMenuItem("Sort");
sort.setActionCommand("Sort Files");
menuSort.setActionCommand("Sort Files");
```

```
JButton ok = new JButton("OK");
ok.requestFocus();
```

```
public void focusGained(FocusEvent event) {
    // ...
}

public void focusLost(FocusEvent event) {
    // ...
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.event.*;
 4: import javax.swing.*;
 5: import java.awt.*;
 6:
 7: public class Calculator extends JFrame implements FocusListener {
 8:     JTextField value1, value2, sum;
 9:     JLabel plus, equals;
10:
11:     public Calculator() {
12:         super("Add Two Numbers");
13:         setSize(350, 90);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         setLookAndFeel();
16:         FlowLayout flow = new FlowLayout(FlowLayout.CENTER);
17:         setLayout(flow);
18:         // create components
19:         value1 = new JTextField("0", 5);
20:         plus = new JLabel("+");
21:         value2 = new JTextField("0", 5);
22:         equals = new JLabel("=");
23:         sum = new JTextField("0", 5);
24:         // add listeners
25:         value1.addFocusListener(this);
26:         value2.addFocusListener(this);
27:         // set up sum field
28:         sum.setEditable(false);
29:         // add components
30:         add(value1);
31:         add(plus);
32:         add(value2);
```

```
33:         add(equals);
34:         add(sum);
35:         setVisible(true);
36:     }
37:
38:     public void focusGained(FocusEvent event) {
39:         try {
40:             float total = Float.parseFloat(value1.getText()) +
41:                 Float.parseFloat(value2.getText());
42:             sum.setText("" + total);
43:         } catch (NumberFormatException nfe) {
44:             value1.setText("0");
45:             value2.setText("0");
46:             sum.setText("0");
47:         }
48:     }
49:
50:     public void focusLost(FocusEvent event) {
51:         focusGained(event);
52:     }
53:
```

```
54:     private void setLookAndFeel() {
55:         try {
56:             UIManager.setLookAndFeel(
57:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
58:             );
59:             SwingUtilities.updateComponentTreeUI(this);
60:         } catch (Exception exc) {
61:             System.err.println("Couldn't use the system "
62:                 + "look and feel: " + exc);
63:         }
64:     }
65:
66:     public static void main(String[] arguments) {
67:         Calculator frame = new Calculator();
68:     }
69: }
```

```
void itemStateChanged(ItemEvent event) {
    // ...
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class FormatChooser extends JFrame implements ItemListener {
 8:     String[] formats = { "(choose format)", "Atom", "RSS 0.92",
 9:         "RSS 1.0", "RSS 2.0" };
10:     String[] descriptions = {
11:         "Atom weblog and syndication format",
12:         "RSS syndication format 0.92 (Netscape)",
13:         "RSS/RDF syndication format 1.0 (RSS/RDF)",
14:         "RSS syndication format 2.0 (UserLand)"
15:     };
16:     JComboBox formatBox = new JComboBox();
17:     JLabel descriptionLabel = new JLabel("");
18:
19:     public FormatChooser() {
20:         super("Syndication Format");
21:         setSize(420, 150);
22:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23:         setLayout(new BorderLayout());
24:         for (int i = 0; i < formats.length; i++) {
25:             formatBox.addItem(formats[i]);
26:         }
27:         formatBox.addItemListener(this);
28:         add(BorderLayout.NORTH, formatBox);
```

```
29:         add(BorderLayout.CENTER, descriptionLabel);
30:         setVisible(true);
31:     }
32:
33:     public void itemStateChanged(ItemEvent event) {
34:         int choice = formatBox.getSelectedIndex();
35:         if (choice > 0) {
36:              descriptionLabel.setText(descriptions[choice-1]);
37:         }
38:     }
39:
40:     public Insets getInsets() {
41:         return new Insets(50, 10, 10, 10);
42:     }
43:
44:     private static void setLookAndFeel() {
45:         try {
46:             UIManager.setLookAndFeel(
47:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
48:             );
49:         } catch (Exception exc) {
50:             System.err.println("Couldn't use the system "
51:                 + "look and feel: " + exc);
52:         }
53:     }
54:
55:     public static void main(String[] arguments) {
56:         FormatChooser.setLookAndFeel();
57:         FormatChooser fc = new FormatChooser();
58:     }
59: }
```

```
public void keyPressed(KeyEvent event) {
    // ...
}

public void keyReleased(KeyEvent event) {
    // ...
}

public void keyTyped(KeyEvent event) {
    // ...
}
```

```
public void mouseReleased(MouseEvent event) {
    // ...
}
```

```
public void mouseDragged(MouseEvent event) {
    // ...
}

public void mouseMoved(MouseEvent event) {
    // ...
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class MousePrank extends JFrame implements ActionListener {
 8:     public MousePrank() {
 9:         super("Message");
10:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11:         setSize(420, 220);
12:         BorderLayout border = new BorderLayout();
13:         setLayout(border);
14:         JLabel msg = new JLabel("Click OK to close program.");
15:         add(BorderLayout.NORTH, msg);
16:         PrankPanel prank = new PrankPanel();
17:         prank.ok.addActionListener(this);
18:         add(BorderLayout.CENTER, prank);
19:         setVisible(true);
20:     }
21:
22:     public void actionPerformed(ActionEvent event) {
23:         System.exit(0);
24:     }
25:
26:     public Insets getInsets() {
27:         return new Insets(40, 10, 10, 10);
28:     }
29:
```

```
30:     private static void setLookAndFeel() {
31:         try {
32:             UIManager.setLookAndFeel(
33:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
34:             );
35:         } catch (Exception exc) {
36:             System.err.println("Couldn't use the system "
37:                 + "look and feel: " + exc);
38:         }
39:     }
40:
41:     public static void main(String[] arguments) {
42:         MousePrank.setLookAndFeel();
43:         new MousePrank();
44:     }
45: }
46:
47: class PrankPanel extends JPanel implements MouseMotionListener {
48:     JButton ok = new JButton("OK");
49:     int buttonX, buttonY, mouseX, mouseY;
50:     int width, height;
51:
52:     PrankPanel() {
53:         super();
54:         setLayout(null);
55:         addMouseMotionListener(this);
56:         buttonX = 110;
```

```
57:         buttonY = 110;
58:         ok.setBounds(new Rectangle(buttonX, buttonY,
59:             70, 20));
60:         add(ok);
61:     }
62:
63:     public void mouseMoved(MouseEvent event) {
64:         mouseX = event.getX();
65:         mouseY = event.getY();
66:         width = (int) getSize().getWidth();
67:         height = (int) getSize().getHeight();
68:         if (Math.abs((mouseX + 35) - buttonX) < 50) {
69:             buttonX = moveButton(mouseX, buttonX, width);
70:             repaint();
71:         }
72:         if (Math.abs((mouseY + 10) - buttonY) < 50) {
73:             buttonY = moveButton(mouseY, buttonY, height);
74:             repaint();
75:         }
76:     }
77:
78:     public void mouseDragged(MouseEvent event) {
79:         // ignore this event
80:     }
```

```
 81:
 82:     private int moveButton(int mouseAt, int buttonAt, int bord) {
 83:         if (buttonAt < mouseAt) {
 84:             buttonAt--;
 85:         } else {
 86:             buttonAt++;
 87:         }
 88:         if (buttonAt > (bord - 20)) {
 89:             buttonAt = 10;
 90:         }
 91:         if (buttonAt < 0) {
 92:             buttonAt = bord - 80;
 93:         }
 94:         return buttonAt;
 95:     }
 96:
 97:     public void paintComponent(Graphics comp) {
 98:         super.paintComponent(comp);
 99:         ok.setBounds(buttonX, buttonY, 70, 20);
100:     }
101: }
```

```
JButton ok = new JButton("OK");
int buttonX = 110;
int buttonY = 110;
ok.setBounds(new Rectangle(buttonX, buttonY, 70, 20));
```

```
Rectangle box = new Rectangle(buttonX, buttonY, 70, 20);
ok.setBounds(box);
```

```
public void windowOpened(WindowEvent event) {
    // body of method
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class KeyChecker extends JFrame {
 8:     JLabel keyLabel = new JLabel("Hit any key");
 9:
10:     public KeyChecker() {
11:         super("Hit a Key");
12:         setSize(300, 200);
13:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:         setLayout(new FlowLayout(FlowLayout.CENTER));
15:         KeyMonitor monitor = new KeyMonitor(this);
16:         setFocusable(true);
17:         addKeyListener(monitor);
18:         add(keyLabel);
19:         setVisible(true);
20:     }
21:
22:     private static void setLookAndFeel() {
```

```
23:         try {
24:             UIManager.setLookAndFeel(
25:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
26:             );
27:         } catch (Exception exc) {
28:             System.err.println("Couldn't use the system "
29:                 + "look and feel: " + exc);
30:         }
31:     }
32:
33:     public static void main(String[] arguments) {
34:         KeyChecker.setLookAndFeel();
35:         new KeyChecker();
36:     }
37: }
38:
39: class KeyMonitor extends KeyAdapter {
40:     KeyChecker display;
41:
42:     KeyMonitor(KeyChecker display) {
43:         this.display = display;
44:     }
45:
46:     public void keyTyped(KeyEvent event) {
47:         display.keyLabel.setText("" + event.getKeyChar());
48:         display.repaint();
49:     }
50: }
```

```
KeyAdapter monitor = new KeyAdapter() {
     public void keyTyped(KeyEvent event) {
          keyLabel.setText("" + event.getKeyChar());
          repaint();
     }
};
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class KeyChecker2 extends JFrame {
 8:     JLabel keyLabel = new JLabel("Hit any key");
 9:
10:     public KeyChecker2() {
11:          super("Hit a Key");
12:          setSize(300, 200);
13:          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14:          setLayout(new FlowLayout(FlowLayout.CENTER));
15:          KeyAdapter monitor = new KeyAdapter() {
16:              public void keyTyped(KeyEvent event) {
17:                   keyLabel.setText("" + event.getKeyChar());
18:                   repaint();
19:              }
20:          };
21:          setFocusable(true);
22:          addKeyListener(monitor);
23:          add(keyLabel);
```

```
24:             setVisible(true);
25:         }
26:
27:         private static void setLookAndFeel() {
28:             try {
29:                 UIManager.setLookAndFeel(
30:                     "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
31:                 );
32:             } catch (Exception exc) {
33:                 System.err.println("Couldn't use the system "
34:                     + "look and feel: " + exc);
35:             }
36:         }
37:
38:         public static void main(String[] arguments) {
39:             KeyChecker2.setLookAndFeel();
40:             new KeyChecker2();
41:         }
42:     }
```

```
JButton b1;
b1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        setTitle("Rosencrantz");
    }
});
JButton b2;
b2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        setTitle("Guildenstern");
    }
});
```

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

public class Expunger extends JFrame implements ActionListener {
    public boolean deleteFile;

    public Expunger() {
        super("Expunger");
        JLabel commandLabel = new JLabel("Do you want to delete the file?");
        JButton yes = new JButton("Yes");
        JButton no = new JButton("No");
        yes.addActionListener(this);
        no.addActionListener(this);
        setLayout( new BorderLayout() );
        JPanel bottom = new JPanel();
        bottom.add(yes);
        bottom.add(no);
        add("North", commandLabel);
        add("South", bottom);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent evt) {
        JButton source = (JButton) evt.getSource();
        // answer goes here
            deleteFile = true;
        else
            deleteFile = false;
    }

    public static void main(String[] arguments) {
        new Expunger();
    }
}
```

```
JFrame main = new JFrame("Welcome Screen");
JPanel pane = new JPanel();
main.add(pane);
```

```
public void paintComponent(Graphics comp) {
    Graphics2D comp2D = (Graphics2D) comp;
    // body of method
}
```

```
public void paintComponent(Graphics comp) {
    Graphics2D comp2D = (Graphics2D) comp;
    comp2D.drawString("Free the bound periodicals", 22, 100);
}
```

```
Font f = new Font("Dialog", Font.BOLD + Font.ITALIC, 24);
```

```
public void paintComponent(Graphics comp) {
    Graphics2D comp2D = (Graphics2D) comp;
    Font f = new Font("Arial Narrow", Font.PLAIN, 72);
    comp2D.setFont(f);
    comp2D.drawString("I'm deeply font of you", 13, 100);
}
```

```
try {
    File ttf = new File("Verdana.ttf");
    FileInputStream fis = new FileInputStream(ttf);
    Font font = Font.createFont(Font.TRUETYPE_FONT, fis);
} catch (IOException|FontFormatException exc) {
    System.out.println("Error: " + exc.getMessage());
}
```

```
comp2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class TextFrame extends JFrame {
 8:     public TextFrame(String text, String fontName) {
 9:         super("Show Font");
10:         setSize(425, 150);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         TextFramePanel sf = new TextFramePanel(text, fontName);
13:         add(sf);
14:         setVisible(true);
15:     }
16:
17:     public static void main(String[] arguments) {
18:         if (arguments.length < 1) {
19:             System.out.println("Usage: java TextFrame msg font");
20:             System.exit(-1);
21:         }
22:         TextFrame tf = new TextFrame(arguments[0], arguments[1]);
23:     }
24:
25: }
26:
```

```
27: class TextFramePanel extends JPanel {
28:     String text;
29:     String fontName;
30:
31:     public TextFramePanel(String text, String fontName) {
32:         super();
33:         this.text = text;
34:         this.fontName = fontName;
35:     }
36:
37:     public void paintComponent(Graphics comp) {
38:         super.paintComponent(comp);
39:         Graphics2D comp2D = (Graphics2D) comp;
40:         comp2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
41:             RenderingHints.VALUE_ANTIALIAS_ON);
42:         Font font = new Font(fontName, Font.BOLD, 18);
43:         FontMetrics metrics = getFontMetrics(font);
44:         comp2D.setFont(font);
45:         int x = (getSize().width - metrics.stringWidth(text)) / 2;
46:         int y = getSize().height / 2;
47:         comp2D.drawString(text, x, y);
48:     }
49: }
```

```
Color c1 = new Color(0.807F, 1F, 0F);

Color c2 = new Color(255, 204, 102);
```

black (0, 0, 0)
blue (0, 0, 255)
cyan (0, 255, 255)
darkGray (64, 64, 64)
gray (128, 128, 128)
green (0, 255, 0)
lightGray (192, 192, 192)
magenta (255, 0, 255)
orange (255, 200, 0)
pink (255, 175, 175)
red (255, 0, 0)
white (255, 255, 255)
yellow (255, 255, 0)

```
Color brush = new Color(255, 204, 102);
comp2D.setColor(brush);
```

```
comp2D.setColor(getBackground());
```

```
GradientPaint gp = new GradientPaint(
    x1, y1, color1, x2, y2, color2);
```

```
GradientPaint gp = new GradientPaint(
    x1, y1, color1, x2, y2, color2, true);
```

```
GradientPaint pat = new GradientPaint(0f, 0f, Color.white,
    100f, 45f, Color.blue);
comp2D.setPaint(pat);
```

```
BasicStroke pen = new BasicStroke(2.0F,
    BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_ROUND);
comp2D.setStroke(pen);
```

```
Line2D.Float ln = new Line2D.Float(60F, 5F, 13F, 28F);
```

```
Rectangle2D.Float rc = new Rectangle2D.Float(10F, 13F, 40F, 20F);
```

```
Ellipse2D.Float ee = new Ellipse2D.Float(113, 25, 22, 40);
```

```
Arc2D.Float arc = new Arc2D.Float(
    27F, 22F, 42F, 30F, 33F, 90F, Arc2D.PIE);
```

```
GeneralPath polly = new GeneralPath();
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.geom.*;
 5: import javax.swing.*;
 6:
 7: public class Map extends JFrame {
 8:     public Map() {
 9:         super("Map");
10:         setSize(360, 350);
11:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         MapPane map = new MapPane();
13:         add(map);
14:         setVisible(true);
15:     }
16:
17:     public static void main(String[] arguments) {
18:         Map frame = new Map();
19:     }
20:
21: }
22:
23: class MapPane extends JPanel {
24:     public void paintComponent(Graphics comp) {
25:         Graphics2D comp2D = (Graphics2D) comp;
26:         comp2D.setColor(Color.blue);
27:         comp2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
28:             RenderingHints.VALUE_ANTIALIAS_ON);
29:         Rectangle2D.Float background = new Rectangle2D.Float(
```

```
30:             0F, 0F, getSize().width, getSize().height);
31:         comp2D.fill(background);
32:         // Draw waves
33:         comp2D.setColor(Color.white);
34:         BasicStroke pen = new BasicStroke(2F,
35:         BasicStroke.CAP_BUTT, BasicStroke.JOIN_ROUND);
36:         comp2D.setStroke(pen);
37:         for (int ax = 0; ax < 340; ax += 10) {
38:             for (int ay = 0; ay < 340 ; ay += 10) {
39:                 Arc2D.Float wave = new Arc2D.Float(ax, ay,
40:                     10, 10, 0, -180, Arc2D.OPEN);
41:                 comp2D.draw(wave);
42:             }
43:         }
44:         // Draw Florida
45:         GradientPaint gp = new GradientPaint(0F, 0F, Color.green,
46:             350F,350F, Color.orange, true);
47:         comp2D.setPaint(gp);
48:         GeneralPath fl = new GeneralPath();
49:         fl.moveTo(10F, 12F);
50:         fl.lineTo(234F, 15F);
51:         fl.lineTo(253F, 25F);
52:         fl.lineTo(261F, 71F);
53:         fl.lineTo(344F, 209F);
54:         fl.lineTo(336F, 278F);
55:         fl.lineTo(295F, 310F);
56:         fl.lineTo(259F, 274F);
57:         fl.lineTo(205F, 188F);
58:         fl.lineTo(211F, 171F);
59:         fl.lineTo(195F, 174F);
60:         fl.lineTo(191F, 118F);
```

```
61:          fl.lineTo(120F, 56F);
62:          fl.lineTo(94F, 68F);
63:          fl.lineTo(81F, 49F);
64:          fl.lineTo(12F, 37F);
65:          fl.closePath();
66:          comp2D.fill(fl);
67:          // Draw ovals
68:          comp2D.setColor(Color.black);
69:          BasicStroke pen2 = new BasicStroke();
70:          comp2D.setStroke(pen2);
71:          Ellipse2D.Float e1 = new Ellipse2D.Float(235, 140, 15, 15);
72:          Ellipse2D.Float e2 = new Ellipse2D.Float(225, 130, 15, 15);
73:           Ellipse2D.Float e3 = new Ellipse2D.Float(245, 130, 15, 15);
74:          comp2D.fill(e1);
75:          comp2D.fill(e2);
76:          comp2D.fill(e3);
77:      }
78: }
```

```
Color c3 = new Color(0xFF, 0xCC, 0x66);
```

```
import java.awt.*;
import javax.swing.*;

public class Result extends JFrame {
    public Result() {
        super("Result");
        JLabel width = new JLabel("This frame is " +
            getSize().width + " pixels wide.");
        add("North", width);
        setSize(220, 120);
    }

    public static void main(String[] arguments) {
        Result r = new Result();
        r.setVisible(true);
    }
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import java.net.*;
 6: import java.io.*;
 7: import javax.swing.*;
 8:
 9: public class PageData extends JFrame implements ActionListener,
10:     Runnable {
11:
12:     Thread runner;
13:     String[] headers = { "Content-Length", "Content-Type",
14:         "Date", "Public", "Expires", "Last-Modified",
15:         "Server" };
16:
17:     URL page;
18:     JTextField url;
19:     JLabel[] headerLabel = new JLabel[7];
20:     JTextField[] header = new JTextField[7];
21:     JButton readPage, clearPage, quitLoading;
22:     JLabel status;
23:
24:     public PageData() {
25:         super("Page Data");
26:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27:         setLookAndFeel();
```

```
28:         setLayout(new GridLayout(10, 1));
29:
30:         JPanel first = new JPanel();
31:         first.setLayout(new FlowLayout(FlowLayout.RIGHT));
32:         JLabel urlLabel = new JLabel("URL:");
33:         url = new JTextField(22);
34:         urlLabel.setLabelFor(url);
35:         first.add(urlLabel);
36:         first.add(url);
37:         add(first);
38:
39:         JPanel second = new JPanel();
40:         second.setLayout(new FlowLayout());
41:         readPage = new JButton("Read Page");
42:         clearPage = new JButton("Clear Fields");
43:         quitLoading = new JButton("Quit Loading");
44:         readPage.setMnemonic('r');
45:         clearPage.setMnemonic('c');
46:         quitLoading.setMnemonic('q');
47:         readPage.setToolTipText("Begin Loading the Web Page");
48:         clearPage.setToolTipText("Clear All Header Fields Below");
49:         quitLoading.setToolTipText("Quit Loading the Web Page");
50:         readPage.setEnabled(true);
51:         clearPage.setEnabled(false);
52:         quitLoading.setEnabled(false);
53:         readPage.addActionListener(this);
```

```
54:         clearPage.addActionListener(this);
55:         quitLoading.addActionListener(this);
56:         second.add(readPage);
57:         second.add(clearPage);
58:         second.add(quitLoading);
59:         add(second);
60:
61:         JPanel[] row = new JPanel[7];
62:         for (int i = 0; i < 7; i++) {
63:             row[i] = new JPanel();
64:             row[i].setLayout(new FlowLayout(FlowLayout.RIGHT));
65:             headerLabel[i] = new JLabel(headers[i] + ":");
66:             header[i] = new JTextField(22);
67:             headerLabel[i].setLabelFor(header[i]);
68:             row[i].add(headerLabel[i]);
69:             row[i].add(header[i]);
70:             add(row[i]);
71:         }
72:
73:         JPanel last = new JPanel();
74:         last.setLayout(new FlowLayout(FlowLayout.LEFT));
75:         status = new JLabel("Enter a URL address to check.");
76:         last.add(status);
77:         add(last);
78:         pack();
79:         setVisible(true);
80:     }
81:
```

```
82:     public void actionPerformed(ActionEvent evt) {
83:         Object source = evt.getSource();
84:         if (source == readPage) {
85:             try {
86:                 page = new URL(url.getText());
87:                 if (runner == null) {
88:                     runner = new Thread(this);
89:                     runner.start();
90:                 }
91:                 quitLoading.setEnabled(true);
92:                 readPage.setEnabled(false);
93:             }
94:             catch (MalformedURLException e) {
95:                 status.setText("Bad URL: " + page);
96:             }
97:         } else if (source == clearPage) {
98:             for (int i = 0; i < 7; i++)
99:                 header[i].setText("");
100:            quitLoading.setEnabled(false);
101:            readPage.setEnabled(true);
102:            clearPage.setEnabled(false);
103:        } else if (source == quitLoading) {
104:            runner = null;
105:            url.setText("");
106:            quitLoading.setEnabled(false);
107:            readPage.setEnabled(true);
108:            clearPage.setEnabled(false);
109:        }
```

```
110:     }
111:
112:     public void run() {
113:         URLConnection conn;
114:         try {
115:             conn = this.page.openConnection();
116:             conn.connect();
117:             status.setText("Connection opened ...");
118:             for (int i = 0; i < 7; i++)
119:                 header[i].setText(conn.getHeaderField(headers[i]));
120:             quitLoading.setEnabled(false);
121:             clearPage.setEnabled(true);
122:             status.setText("Done");
123:             runner = null;
124:         }
125:         catch (IOException e) {
126:             status.setText("IO Error:" + e.getMessage());
127:         }
128:     }
129:
130:     private static void setLookAndFeel() {
131:         try {
132:             UIManager.setLookAndFeel(
133:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
134:             );
135:         } catch (Exception exc) {
136:             // ignore error
137:         }
138:     }
139:
140:
141:     public static void main(String[] arguments) {
142:         PageData frame = new PageData();
143:     }
144: }
```

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <!-- JNLP File for the PageData Application -->
 3: <jnlp
 4:   codebase="http://cadenhead.org/book/java-21-days/java"
 5:   href="PageData.jnlp">
 6:   <information>
 7:     <title>PageData Application</title>
 8:     <vendor>Rogers Cadenhead</vendor>
 9:     <homepage href="http://www.java21days.com"/>
10:     <icon href="pagedataicon.gif"/>
11:     <offline-allowed/>
12:   </information>
13:   <resources>
14:     <j2se version="1.8"/>
15:     <jar href="PageData.jar"/>
16:   </resources>
17:   <security>
18:     <j2ee-application-client-permissions/>
19:   </security>
20:   <application-desc main-class="PageData"/>
21: </jnlp>
```

```
<title>PageData Application</title>
```

```
<homepage href="http://www.java21days.com"/>
```

```
<security>
  <j2ee-application-client-permissions/>
</security>
```

http://cadenhead.org/book/java-21-days/java/PageData.jnlp

http://cadenhead.org/book/java21days/java/pagedataicon.gif

```
<application-desc main-class="PageData">
  <argument>http://java.com</argument>
  <argument>yes</argument>
</application-desc>
```

application/x-java-jnlp-file JNLP

```
<security>
  <j2ee-application-client-permission/>
</security>
```

```
<description>The PageData application.</description>
<description kind="one-line">An application to learn more about web
servers and pages.</description>
<description kind="tooltip">Learn about web servers and
pages.</description>
<description kind="short">PageData, a simple Java application that
takes a URL and displays information about the URL and the web
server that delivered it.</description>
```

```
<icon kind="splash" href="pagedatasplash.gif" width="300"
height="200" />
```

```
public class DiceWorker extends SwingWorker {
    // body of class
}
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4:
 5: public class DiceWorker extends SwingWorker {
 6:     int timesToRoll;
 7:
 8:     // set up the Swing worker
 9:     public DiceWorker(int timesToRoll) {
10:         super();
11:         this.timesToRoll = timesToRoll;
12:     }
13:
14:     // define the task the worker performs
15:     protected int[] doInBackground() {
16:         int[] result = new int[16];
17:         for (int i = 0; i < this.timesToRoll; i++) {
18:             int sum = 0;
19:             for (int j = 0; j < 3; j++) {
20:                 sum += Math.floor(Math.random() * 6);
21:             }
22:             result[sum] = result[sum] + 1;
23:         }
24:         // transmit the result
25:         return result;
26:     }
27: }
```

```
public void propertyChange(PropertyChangeEvent event) {
    // ...
}
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import java.beans.*;
 6: import javax.swing.*;
 7:
 8: public class DiceRoller extends JFrame implements ActionListener,
 9:     PropertyChangeListener {
10:
11:     // the table for dice-roll results
12:     JTextField[] total = new JTextField[16];
13:     // the "Roll" button
14:     JButton roll;
15:     // the number of times to roll
16:     JTextField quantity;
17:     // the Swing worker
18:     DiceWorker worker;
19:
20:     public DiceRoller() {
21:         super("Dice Roller");
22:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23:         setLookAndFeel();
24:         setSize(850, 145);
25:
```

```
26:         // set up top row
27:         JPanel topPane = new JPanel();
28:         GridLayout paneGrid = new GridLayout(1, 16);
29:         topPane.setLayout(paneGrid);
30:         for (int i = 0; i < 16; i++) {
31:             // create a textfield and label
32:             total[i] = new JTextField("0", 4);
33:             JLabel label = new JLabel((i + 3) + ": ");
34:             // create this cell in the grid
35:             JPanel cell = new JPanel();
36:             cell.add(label);
37:             cell.add(total[i]);
38:             // add the cell to the top row
39:             topPane.add(cell);
40:         }
41:
42:         // set up bottom row
43:         JPanel bottomPane = new JPanel();
44:         JLabel quantityLabel = new JLabel("Times to Roll: ");
45:         quantity = new JTextField("0", 5);
46:         roll = new JButton("Roll");
47:         roll.addActionListener(this);
48:         bottomPane.add(quantityLabel);
49:         bottomPane.add(quantity);
50:         bottomPane.add(roll);
51:
52:         // set up frame
53:         GridLayout frameGrid = new GridLayout(2, 1);
54:         setLayout(frameGrid);
```

```
55:         add(topPane);
56:         add(bottomPane);
57:
58:         setVisible(true);
59:     }
60:
61:     // respond when the "Roll" button is clicked
62:     public void actionPerformed(ActionEvent event) {
63:         int timesToRoll;
64:         try {
65:             // turn off the button
66:             timesToRoll = Integer.parseInt(quantity.getText());
67:             roll.setEnabled(false);
68:             // set up the worker that will roll the dice
69:             worker = new DiceWorker(timesToRoll);
70:             // add a listener that monitors the worker
71:             worker.addPropertyChangeListener(this);
72:             // start the worker
73:             worker.execute();
74:         } catch (Exception exc) {
75:             System.out.println(exc.getMessage());
76:             exc.printStackTrace();
77:         }
78:     }
79:
```

```
80:     // respond when the worker's task is complete
81:     public void propertyChange(PropertyChangeEvent event) {
82:         try {
83:             // get the worker's dice-roll results
84:             int[] result = (int[]) worker.get();
85:             // store the results in text fields
86:             for (int i = 0; i < result.length; i++) {
87:                 total[i].setText("" + result[i]);
88:             }
89:         } catch (Exception exc) {
90:             System.out.println(exc.getMessage());
91:             exc.printStackTrace();
92:         }
93:     }
94:
95:     private static void setLookAndFeel() {
96:         try {
97:             UIManager.setLookAndFeel(
98:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
99:             );
100:        } catch (Exception exc) {
101:            // ignore error
102:        }
103:    }
104:
105:    public static void main(String[] arguments) {
106:        new DiceRoller();
107:    }
108: }
```

```
int[] result = (int[] worker.get();
```

```
import java.awt.*;
import javax.swing.*;

public class SliderFrame extends JFrame {
    public SliderFrame() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JSlider value = new JSlider(0, 255, 100);
        setSize(325, 150);
        setVisible(true);
    }

    public static void main(String[] arguments) {
        new SliderFrame();
    }
}
```

```
FileInputStream fis = new FileInputStream("scores.dat");
```

```
FileInputStream f1 = new FileInputStream("C:\\data\\calendar.txt");
```

```
FileInputStream f2 = new FileInputStream("/data/calendar.txt");
```

```
char sep = File.separator;
FileInputStream f2 = new FileInputStream(sep + "data"
    + sep + "calendar.txt");
```

```
int newByte = 0;
while (newByte != -1) {
    newByte = diskfile.read();
    System.out.print(newByte + " ");
}
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class ByteReader {
 6:     public static void main(String[] arguments) {
 7:         try (
 8:             FileInputStream file = new
 9:                 FileInputStream("save.gif")
10:             ) {
11:
12:             boolean eof = false;
13:             int count = 0;
14:             while (!eof) {
15:                 int input = file.read();
16:                 System.out.print(input + " ");
17:                 if (input == -1)
18:                     eof = true;
19:                 else
20:                     count++;
21:             }
22:             file.close();
23:             System.out.println("\nBytes read: " + count);
24:         } catch (IOException e) {
25:             System.out.println("Error -- " + e.toString());
26:         }
27:     }
28: }
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class ByteWriter {
 6:     public static void main(String[] arguments) {
 7:         int[] data = { 71, 73, 70, 56, 57, 97, 13, 0, 12, 0, 145,
 8:             0, 0, 255, 255, 255, 255, 255, 0, 0, 0, 0, 0, 0, 0, 44,
 9:             0, 0, 0, 0, 13, 0, 12, 0, 0, 2, 38, 132, 45, 121, 11,
10:             25, 175, 150, 120, 20, 162, 132, 51, 110, 106, 239, 22,
11:             8, 160, 56, 137, 96, 72, 77, 33, 130, 86, 37, 219, 182,
12:             230, 137, 89, 82, 181, 50, 220, 103, 20, 0, 59 };
13:         try (FileOutputStream file = new
14:             FileOutputStream("pic.gif")) {
15:
16:             for (int i = 0; i < data.length; i++) {
17:                 file.write(data[i]);
18:             }
19:             file.close();
20:         } catch (IOException e) {
21:             System.out.println("Error -- " + e.toString());
22:         }
23:     }
24: }
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class BufferDemo {
 6:     public static void main(String[] arguments) {
 7:         int start = 0;
 8:         int finish = 255;
 9:         if (arguments.length > 1) {
10:             start = Integer.parseInt(arguments[0]);
11:             finish = Integer.parseInt(arguments[1]);
12:         } else if (arguments.length > 0) {
13:             start = Integer.parseInt(arguments[0]);
14:         }
15:         ArgStream as = new ArgStream(start, finish);
16:         System.out.println("\nWriting: ");
17:         boolean success = as.writeStream();
18:         System.out.println("\nReading: ");
19:         boolean readSuccess = as.readStream();
20:     }
21: }
22:
23: class ArgStream {
24:     int start = 0;
25:     int finish = 255;
26:
27:     ArgStream(int st, int fin) {
28:         start = st;
29:         finish = fin;
30:     }
31:
```

```
32:     boolean writeStream() {
33:         try (FileOutputStream file = new
34:                 FileOutputStream("numbers.dat");
35:             BufferedOutputStream buff = new
36:                 BufferedOutputStream(file)) {
37:
38:             for (int out = start; out <= finish; out++) {
39:                 buff.write(out);
40:                 System.out.print(" " + out);
41:             }
42:             buff.close();
43:             return true;
44:         } catch (IOException e) {
45:             System.out.println("Exception: " + e.getMessage());
46:             return false;
47:         }
48:     }
49:
50:     boolean readStream() {
51:         try (FileInputStream file = new
52:                 FileInputStream("numbers.dat");
```

```
53:             BufferedInputStream buff = new
54:                 BufferedInputStream(file)) {
55:
56:             int in;
57:             do {
58:                 in = buff.read();
59:                 if (in != -1) {
60:                     System.out.print(" " + in);
61:                 }
62:             } while (in != -1);
63:             System.out.println();
63:             buff.close();
64:             return true;
65:         } catch (IOException e) {
66:             System.out.println("Exception: " + e.getMessage());
67:             return false;
68:         }
69:     }
70: }
```

```
BufferedInputStream command = new BufferedInputStream(System.in);
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class ConsoleInput {
 6:     public static String readLine() {
 7:         StringBuilder response = new StringBuilder();
 8:         try (BufferedInputStream buff = new
 9:             BufferedInputStream(System.in)) {
10:
11:             int in;
12:             char inChar;
13:             do {
14:                 in = buff.read();
15:                 inChar = (char) in;
16:                 if ((in != -1) & (in != '\n') & (in != '\r')) {
17:                     response.append(inChar);
18:                 }
19:             } while ((in != -1) & (inChar != '\n') & (in != '\r'));
20:             buff.close();
21:             return response.toString();
22:         } catch (IOException e) {
23:             System.out.println("Exception: " + e.getMessage());
24:             return null;
25:         }
26:     }
27:
28:     public static void main(String[] arguments) {
29:         System.out.print("\nWhat is your name? ");
30:         String input = ConsoleInput.readLine();
31:         System.out.println("\nHello, " + input);
32:     }
33: }
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class PrimeWriter {
 6:     public static void main(String[] arguments) {
 7:         int[] primes = new int[400];
 8:         int numPrimes = 0;
 9:         // candidate: the number that might be prime
10:         int candidate = 2;
11:         while (numPrimes < 400) {
12:             if (isPrime(candidate)) {
13:                 primes[numPrimes] = candidate;
14:                 numPrimes++;
15:             }
16:             candidate++;
17:         }
18:
19:         try (
20:             // Write output to disk
21:             FileOutputStream file = new
22:                 FileOutputStream("400primes.dat");
23:             BufferedOutputStream buff = new
```

```
24:                 BufferedOutputStream(file);
25:             DataOutputStream data = new
26:                 DataOutputStream(buff);
27:             ) {
28:
29:             for (int i = 0; i < 400; i++)
30:                data.writeInt(primes[i]);
31:             data.close();
32:         } catch (IOException e) {
33:             System.out.println("Error -- " + e.toString());
34:         }
35:     }
36:
37:     public static boolean isPrime(int checkNumber) {
38:         double root = Math.sqrt(checkNumber);
39:         for (int i = 2; i <= root; i++) {
40:             if (checkNumber % i == 0)
41:                 return false;
42:         }
43:         return true;
44:     }
45: }
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class PrimeReader {
 6:     public static void main(String[] arguments) {
 7:         try (FileInputStream file = new
 8:                 FileInputStream("400primes.dat");
 9:             BufferedInputStream buff = new
10:                 BufferedInputStream(file);
11:             DataInputStream data = new
12:                 DataInputStream(buff)) {
13:
14:             try {
15:                 while (true) {
16:                     int in = data.readInt();
17:                     System.out.print(in + " ");
18:                 }
19:             } catch (EOFException eof) {
20:                 buff.close();
21:             }
22:         } catch (IOException e) {
23:             System.out.println("Error -- " + e.toString());
24:         }
25:     }
26: }
```

```
FileReader look = new FileReader("index.txt");
```

```
FileReader text = new FileReader("readme.txt");
int inByte;
do {
    inByte = text.read();
    if (inByte != -1) {
        System.out.print( (char) inByte );
    }
} while (inByte != -1);
System.out.println("");
text.close();
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4:
 5: public class SourceReader {
 6:     public static void main(String[] arguments) {
 7:         try (
 8:             FileReader file = new
 9:                 FileReader("SourceReader.java");
10:             BufferedReader buff = new
11:                 BufferedReader(file)) {
12:
13:             boolean eof = false;
14:             while (!eof) {
15:                 String line = buff.readLine();
16:                 if (line == null) {
17:                     eof = true;
18:                 } else {
19:                     System.out.println(line);
20:                 }
21:             }
22:             buff.close();
23:         } catch (IOException e) {
24:             System.out.println("Error -- " + e.toString());
25:         }
26:     }
27: }
```

```
FileWriter letters = new FileWriter("alphabet.txt");
for (int i = 65; i < 91; i++)
    letters.write( (char) i );
letters.close();
```

```
Path source = FileSystems.getDefault().getPath("essay.txt");
```

```
File sourceFile = source.toFile();
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.nio.file.*;
 5:
 6: public class AllCapsDemo {
 7:     public static void main(String[] arguments) {
 8:         if (arguments.length < 1) {
 9:             System.out.println("You must specify a filename");
10:             System.exit(-1);
11:         }
12:         AllCaps cap = new AllCaps(arguments[0]);
13:         cap.convert();
14:     }
15: }
16:
17: class AllCaps {
18:     String sourceName;
19:
20:     AllCaps(String sourceArg) {
21:         sourceName = sourceArg;
22:     }
23:
24:     void convert() {
25:         try {
26:             // Create file objects
27:             FileSystem fs = FileSystems.getDefault();
28:             Path source = fs.getPath(sourceName);
29:             Path temp = fs.getPath("tmp_" + sourceName);
30:
31:             // Create input stream
32:             FileReader fr = new FileReader(source.toFile());
33:             BufferedReader in = new BufferedReader(fr);
```

```
34:
35:             // Create output stream
36:             FileWriter fw = new FileWriter(temp.toFile());
37:             BufferedWriter out = new
38:                 BufferedWriter(fw);
39:
40:             boolean eof = false;
41:             int inChar;
42:             do {
43:                 inChar = in.read();
44:                 if (inChar != -1) {
45:                     char outChar = Character.toUpperCase(
46:                         (char) inChar);
47:                     out.write(outChar);
48:                 } else
49:                     eof = true;
50:             } while (!eof);
51:             in.close();
52:             out.close();
53:
54:             Files.delete(source);
55:             Files.move(temp, source);
56:         } catch (IOException|SecurityException se) {
57:             System.out.println("Error -- " + se.toString());
58:         }
59:     }
60: }
```

```
String userFolder = System.getProperty("user.dir");
```

```
import java.io.*;

public class Unknown {
    public static void main(String[] arguments) {
        String command = "";
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
        try {
            command = br.readLine();
        }
        catch (IOException e) { }
    }
}
```

```
public class Hello {

    class InnerHello {
        InnerHello() {
            System.out.println(
                "The method call is coming from inside the class!"
             );
        }
    }

    public Hello() {
        // empty constructor
    }

    public static void main(String[] arguments) {
        Hello program = new Hello();
        Hello.InnerHello inner = program.new InnerHello();
    }
}
```

```
Hello.InnerHello inner = program.new InnerHello();
```

```
 1: package com.java21days;
 2:
 3: import java.util.*;
 4:
 5: public class ComicBox {
 6:     class InnerComic {
 7:         String title;
 8:         String issueNumber;
 9:         String condition;
10:         float basePrice;
11:         float price;
12:
13:         InnerComic(String inTitle, String inIssueNumber,
14:             String inCondition, float inBasePrice) {
15:
16:             title = inTitle;
17:             issueNumber = inIssueNumber;
18:             condition = inCondition;
19:             basePrice = inBasePrice;
20:         }
21:
22:         void setPrice(float factor) {
23:             price = basePrice * factor;
24:         }
25:     }
26:
27:     public ComicBox() {
28:         HashMap<String, Float> quality = new HashMap<>();
29:         float price1 = 3.00F;
30:         quality.put("mint", price1);
```

```
31:         float price2 = 2.00F;
32:         quality.put("near mint", price2);
33:         float price3 = 1.50F;
34:         quality.put("very fine", price3);
35:         float price4 = 1.00F;
36:         quality.put("fine", price4);
37:         float price5 = 0.50F;
38:         quality.put("good", price5);
39:         float price6 = 0.25F;
40:         quality.put("poor", price6);
41:         InnerComic[] comix = new InnerComic[3];
42:         comix[0] = new InnerComic("Amazing Spider-Man", "1A",
43:             "very fine", 12_000.00F);
44:         comix[0].setPrice(quality.get(comix[0].condition));
45:         comix[1] = new InnerComic("Incredible Hulk", "181",
46:             "near mint", 680.00F);
47:         comix[1].setPrice(quality.get(comix[1].condition));
48:         comix[2] = new InnerComic("Cerebus", "1A", "good", 190.00F);
49:         comix[2].setPrice(quality.get(comix[2].condition));
50:         for (InnerComic comix1 : comix) {
51:             System.out.println("Title: " + comix1.title);
52:             System.out.println("Issue: " + comix1.issueNumber);
53:             System.out.println("Condition: " + comix1.condition);
54:             System.out.println("Price: $" + comix1.price + "\n");
55:         }
56:     }
57:
58:     public static void main(String[] arguments) {
59:         new ComicBox();
60:     }
61: }
```

```
ComicBox.InnerComic[] comix = new ComicBox.InnerComic[3];
```

```
ThreadClass task = new ThreadClass();
Thread runner = new Thread(task);
runner.start();
```

```
Thread runner = new Thread(new Runnable()   {
    public void run() {
        // thread does its work here
    }
});
runner.start();
```

```
new Runnable()   {
     public void run() {
          // thread does its work here
     }
)
```

```
public void windowOpened(WindowEvent event) {
    Window pane = event.getWindow();
    pane.setBackground(Color.CYAN);
}

public void windowClosed(WindowEvent event) {
    // do nothing
}

public void windowActivated(WindowEvent event) {
    // do nothing
}

public void windowDeactivated(WindowEvent event) {
    // do nothing
}
```

```
public class WindowCloseListener extends WindowAdapter {
    JFrame frame;
    boolean done;

    public WindowCloseListener(JFrame inFrame) {
        this.frame = inFrame;
    }

    public void windowClosing(WindowEvent event) {
        // user has tried to close window
        if (frame.done) {
            // allow it
            frame.dispose();
            System.exit(0);
        }
    }
}
```

```
setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

```
WindowCloseListener closer = new WindowCloseListener();
addWindowListener(closer);
```

```
setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
addWindowListener(new WindowAdapter() {
    // user has tried to close window
    if (frame.done) {
        // allow it
        frame.dispose();
        System.exit(0);
    }
});
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class ProgressMonitor2 extends JFrame {
 8:     JProgressBar current;
 9:     int num = 0;
10:     boolean done = false;
11:
12:     public ProgressMonitor2() {
13:         super("Progress Monitor 2");
14:         setLookAndFeel();
15:         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
16:         addWindowListener(new WindowAdapter() {
17:             @Override
18:             public void windowClosing(WindowEvent event) {
19:                 // user has tried to close window
20:                 if (done) {
21:                     // allow it
22:                     dispose();
23:                     System.exit(0);
24:                 }
25:             }
26:         });
27:         setSize(400, 100);
28:         setLayout(new FlowLayout());
29:         current = new JProgressBar(0, 2000);
```

```
30:         current.setValue(0);
31:         current.setStringPainted(true);
32:         current.setPreferredSize(new Dimension(360, 48));
33:         add(current);
34:         setVisible(true);
35:         iterate();
36:     }
37:
38:     public final void iterate() {
39:         while (num < 2000) {
40:             current.setValue(num);
41:             try {
42:                 Thread.sleep(1000);
43:             } catch (InterruptedException e) { }
44:             num += 95;
45:         }
46:         done = true;
47:     }
48:
49:     private void setLookAndFeel() {
50:         try {
51:             UIManager.setLookAndFeel(
52:                 "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
53:             );
```

```
54:                 SwingUtilities.updateComponentTreeUI(this);
55:             } catch (Exception e) {
56:                 System.err.println("Couldn't use the system "
57:                     + "look and feel: " + e);
58:             }
59:         }
60:
61:         public static void main(String[] arguments) {
62:             new ProgressMonitor2();
63:         }
64: }
```

```
Runnable runner = () -> { System.out.println("Eureka!"); };
```

```
public void run() {
    System.out.println("Eureka!");
}
```

```
ActionListener listen = (ActionEvent act) -> {
    System.out.println(act.getSource());
};
```

```
public void actionPerformed(ActionEvent act) {
    System.out.println(act.getSource());
}
```

```
ActionListener listen = (act) -> {
    System.out.println(act.getSource());
}
```

```
JPanel panel = new JPanel();
panel.setCursor(new Cursor(Cursor.TEXT_CURSOR));
```

```
panel.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class CursorMayhem extends JFrame {
 8:     JButton harry, wade, hansel;
 9:
10:     public CursorMayhem() {
11:         super("Choose a Cursor");
12:         setLookAndFeel();
13:         setSize(400, 80);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         setLayout(new FlowLayout());
16:         harry = new JButton("Crosshair");
17:         add(harry);
18:         wade = new JButton("Wait");
19:         add(wade);
20:         hansel = new JButton("Hand");
21:         add(hansel);
22:         // begin anonymous inner class
23:         ActionListener act = new ActionListener() {
24:             public void actionPerformed(ActionEvent event) {
25:                 if (event.getSource() == harry) {
26:                     setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
27:                 }
28:                 if (event.getSource() == wade) {
29:                     setCursor(new Cursor(Cursor.WAIT_CURSOR));
```

```
30:                     }
31:                     if (event.getSource() == hansel) {
32:                         setCursor(new Cursor(Cursor.HAND_CURSOR));
33:                     }
34:                 }
35:             };
36:             // end anonymous inner class
37:             harry.addActionListener(act);
38:             wade.addActionListener(act);
39:             hansel.addActionListener(act);
40:             setVisible(true);
41:         }
42:
43:         private void setLookAndFeel() {
44:             try {
45:                 UIManager.setLookAndFeel(
46:                     "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
47:                 );
48:             } catch (Exception exc) {
49:                 System.err.println("Look and feel error: " + exc);
50:             }
51:         }
52:
53:         public static void main(String[] arguments) {
54:             new CursorMayhem();
55:         }
56: }
```

```
 1: package com.java21days;
 2:
 3: import java.awt.*;
 4: import java.awt.event.*;
 5: import javax.swing.*;
 6:
 7: public class ClosureMayhem extends JFrame {
 8:     JButton harry, wade, hansel;
 9:
10:     public ClosureMayhem() {
11:         super("Choose a Cursor");
12:         setLookAndFeel();
13:         setSize(400, 80);
14:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         setLayout(new FlowLayout());
16:         harry = new JButton("Crosshair");
17:         add(harry);
18:         wade = new JButton("Wait");
19:         add(wade);
20:         hansel = new JButton("Hand");
21:         add(hansel);
22:         // begin closure
23:         ActionListener act = (event) -> {
24:             if (event.getSource() == harry) {
25:                 setCursor(new Cursor(Cursor.CROSSHAIR_CURSOR));
26:             }
27:             if (event.getSource() == wade) {
28:                 setCursor(new Cursor(Cursor.WAIT_CURSOR));
```

```
29:                 }
30:                 if (event.getSource() == hansel) {
31:                     setCursor(new Cursor(Cursor.HAND_CURSOR));
32:                 }
33:             };
34:             // end closure
35:             harry.addActionListener(act);
36:             wade.addActionListener(act);
37:             hansel.addActionListener(act);
38:             setVisible(true);
39:         }
40:
41:         private void setLookAndFeel() {
42:             try {
43:                 UIManager.setLookAndFeel(
44:                     "com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"
45:                 );
46:             } catch (Exception exc) {
47:                 System.err.println("Look and feel error: " + exc);
48:             }
49:         }
51:
52:         public static void main(String[] arguments) {
53:             new ClosureMayhem();
54:         }
55: }
```

```
public class ClassType {
    public static void main(String[] arguments) {
        Class c = String.class;
        try {
            Object o = c.newInstance();
            if (o instanceof String) {
                System.out.println("True");
            } else {
                System.out.println("False");
            }
        } catch (Exception e) {
            System.out.println("Error");
        }
    }
}
```

```
try {
    URL load = new URL("http://www.samspublishing.com");
} catch (MalformedURLException e) {
    System.out.println("Bad URL");
}
```

```
 1: package com.java21days;
 2:
 3: import javax.swing.*;
 4: import java.net.*;
 5: import java.io.*;
 6:
 7: public class WebReader extends JFrame {
 8:     JTextArea box = new JTextArea("Getting data ...");
 9:
10:     public WebReader() {
11:         super("Get File Application");
12:         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13:         setSize(600, 300);
14:         JScrollPane pane = new JScrollPane(box);
15:         add(pane);
16:         setVisible(true);
17:     }
18:
19:     void getData(String address) throws MalformedURLException {
20:         setTitle(address);
21:         URL page = new URL(address);
22:         StringBuilder text = new StringBuilder();
23:         try {
24:             HttpURLConnection conn = (HttpURLConnection)
25:                 page.openConnection();
26:             conn.connect();
27:             InputStreamReader in = new InputStreamReader(
28:                 (InputStream) conn.getContent());
```

```
29:             BufferedReader buff = new BufferedReader(in);
30:             box.setText("Getting data ...");
31:             String line;
32:             do {
33:                 line = buff.readLine();
34:                 text.append(line);
35:                 text.append("\n");
36:             } while (line != null);
37:             box.setText(text.toString());
38:         } catch (IOException ioe) {
39:             System.out.println("IO Error:" + ioe.getMessage());
40:         }
41:     }
42:
43:     public static void main(String[] arguments) {
44:         if (arguments.length < 1) {
45:             System.out.println("Usage: java WebReader url");
46:             System.exit(1);
47:         }
48:         try {
49:             WebReader app = new WebReader();
50:             app.getData(arguments[0]);
51:         } catch (MalformedURLException mue) {
52:             System.out.println("Bad URL: " + arguments[0]);
53:         }
54:     }
55: }
```

```
String key;
String header;
int i = 0;
do {
      key = conn.getHeaderFieldKey(i);
      header = conn.getHeaderField(i);
      if (key == null) {
            key = "";
      } else {
            key = key + ": ";
      }
      if (header != null) {
            text.append(key);
            text.append(header);
            text.append("\n");
      }
      i++;
} while (header != null);
text.append("\n");
```

```
Socket connection = new Socket(hostName, portNumber);
```

```
connection.setSoTimeOut(50000);
```

```
BufferedInputStream bis = new
    BufferedInputStream(connection.getInputStream());
DataInputStream in = new DataInputStream(bis);

BufferedOutputStream bos = new
    BufferedOutputStream(connection.getOutputStream());
DataOutputStream out = new DataOutputStream(bos);
```

```
DataInputStream in = new DataInputStream(
    new BufferedInputStream(
        sock.getInputStream()));
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.util.*;
 6:
 7: public class Finger {
 8:     public static void main(String[] args) {
 9:         String user;
10:         String host;
11:         if ((args.length == 1) && (args[0].indexOf("@") > -1)) {
12:             StringTokenizer split = new StringTokenizer(args[0],
13:                 "@");
14:             user = split.nextToken();
15:             host = split.nextToken();
16:         } else {
17:             System.out.println("Usage: java Finger user@host");
18:             return;
19:         }
20:         try (Socket digit = new Socket(host, 79);
21:             BufferedReader in = new BufferedReader(
22:                 new InputStreamReader(digit.getInputStream()));
23:             ) {
24:
25:             digit.setSoTimeout(20000);
26:             PrintStream out = new PrintStream(
27:                 digit.getOutputStream());
28:             out.print(user + "\015\012");
29:
```

```
30:             boolean eof = false;
31:             while (!eof) {
32:                 String line = in.readLine();
33:                 if (line != null) {
34:                     System.out.println(line);
35:                 } else {
36:                     eof = true;
37:                 }
38:             }
39:             digit.close();
40:         } catch (IOException e) {
41:             System.out.println("IO Error:" + e.getMessage());
42:         }
43:     }
44: }
```

```
ServerSocket servo = new ServerSocket(8888);
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.util.*;
 6:
 7: public class TimeServer extends Thread {
 8:     private ServerSocket sock;
 9:
10:     public TimeServer() {
11:         super();
12:         try {
13:             sock = new ServerSocket(4415);
14:             System.out.println("TimeServer running ...");
15:         } catch (IOException e) {
16:             System.out.println("Error: couldn't create socket.");
17:             System.exit(1);
18:         }
19:     }
20:
21:     public void run() {
22:         Socket client = null;
23:
24:         while (true) {
25:             if (sock == null)
26:                 return;
27:             try {
```

```
28:                     client = sock.accept();
29:                     BufferedOutputStream bb = new BufferedOutputStream(
30:                         client.getOutputStream());
31:                     PrintWriter os = new PrintWriter(bb, false);
32:                     String outLine;
33:
34:                     Date now = new Date();
35:                     os.println(now);
36:                     os.flush();
37:
38:                     os.close();
39:                     client.close();
40:                 } catch (IOException e) {
41:                     System.out.println("Error: couldn't connect.");
42:                     System.exit(1);
43:                 }
44:             }
45:         }
46:
47:         public static void main(String[] arguments) {
48:             TimeServer server = new TimeServer();
49:             server.start();
50:         }
51:
52: }
```

```
int[] temperatures = { 90, 85, 87, 78, 80, 75, 70, 79, 85, 92 };
IntBuffer tempBuffer = IntBuffer.wrap(temperatures);
```

```
for (int i = 0; tempBuffer.remaining() > 0; i++)
    System.out.println(tempBuffer.get());
```

```
int[] temps = { 90, 85, 87, 78, 80, 75, 70, 79, 85, 92, 99 };
IntBuffer tempBuffer = IntBuffer.allocate(temps.length);
for (int i = 0; i < temps.length; i++) {
    float celsius = ( (float) temps[i] - 32 ) / 9 * 5;
    tempBuffer.put( (int) celsius );
}
tempBuffer.position(0);
for (int i = 0; tempBuffer.remaining() > 0; i++) {
    System.out.println(tempBuffer.get());
}
```

```
Charset isoset = Charset.forName("ISO-8859-1");
```

```
ByteBuffer netBuffer = ByteBuffer.allocate(20480);
// code to fill byte buffer would be here
Charset set = Charset.forName("ISO-8859-1");
CharsetDecoder decoder = set.newDecoder();
netBuffer.position(0);
CharBuffer netText = decoder.decode(netBuffer);
```

```
try {
    String source = "prices.dat";
    FileInputStream inSource = new FileInputStream(source);
    FileChannel inChannel = inSource.getChannel();
} catch (FileNotFoundException fne) {
    System.out.println(fne.getMessage());
}
```

```
long inSize = inChannel.size();
ByteBuffer data = ByteBuffer.allocate( (int) inSize );
inChannel.read(data, 0);
data.position(0);
for (int i = 0; data.remaining() > 0; i++) {
    System.out.print(data.get() + " ");
}
```

```
 1: package com.java21days;
 2:
 3: import java.nio.*;
 4: import java.nio.channels.*;
 5: import java.nio.charset.*;
 6: import java.io.*;
 7:
 8: public class BufferConverter {
 9:     public static void main(String[] arguments) {
10:         try {
11:             // read byte data into a byte buffer
12:             String data = "friends.dat";
13:             FileInputStream inData = new FileInputStream(data);
14:             FileChannel inChannel = inData.getChannel();
15:             long inSize = inChannel.size();
16:             ByteBuffer source = ByteBuffer.allocate((int) inSize);
17:             inChannel.read(source, 0);
18:             source.position(0);
19:             System.out.println("Original byte data:");
20:             for (int i = 0; source.remaining() > 0; i++) {
21:                 System.out.print(source.get() + " ");
22:             }
23:             // convert byte data into character data
24:             source.position(0);
25:             Charset ascii = Charset.forName("US-ASCII");
```

```
26:             CharsetDecoder toAscii = ascii.newDecoder();
27:             CharBuffer destination = toAscii.decode(source);
28:             destination.position(0);
29:             System.out.println("\n\nNew character data:");
30:             for (int i = 0; destination.remaining() > 0; i++) {
31:                 System.out.print(destination.get());
32:             }
33:             System.out.println();
34:         } catch (FileNotFoundException fne) {
35:             System.out.println(fne.getMessage());
36:         } catch (IOException ioe) {
37:             System.out.println(ioe.getMessage());
38:         }
39:     }
40: }
```

```
Selector monitor = Selector.open();
```

```
Selector spy = Selector.open();
channel.register(spy, SelectionKey.OP_READ, null);
```

```
Selector spy = Selector.open();
channel.register(spy, SelectionKey.OP_READ + SelectionKey.OP_WRITE,
    null);
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.nio.channels.*;
 6: import java.util.*;
 7:
 8: public class FingerServer {
 9:
10:     public FingerServer() {
11:         try {
12:             // Create a non-blocking server socket channel
13:             ServerSocketChannel sock = ServerSocketChannel.open();
14:             sock.configureBlocking(false);
15:
16:             // Set the host and port to monitor
17:             InetSocketAddress server = new InetSocketAddress(
18:                 "localhost", 79);
19:             ServerSocket socket = sock.socket();
20:             socket.bind(server);
21:
22:             // Create the selector and register it on the channel
23:             Selector selector = Selector.open();
24:             sock.register(selector, SelectionKey.OP_ACCEPT);
25:
26:             // Loop forever, looking for client connections
27:             while (true) {
28:                 // Wait for a connection
```

```
29:                 selector.select();
30:
31:                 // Get list of selection keys with pending events
32:                 Set keys = selector.selectedKeys();
33:                 Iterator it = keys.iterator();
34:
35:                 // Handle each key
36:                 while (it.hasNext()) {
37:
38:                     // Get the key and remove it from the iteration
39:                     SelectionKey sKey = (SelectionKey) it.next();
40:
41:                     it.remove();
42:                     if (sKey.isAcceptable()) {
43:
44:                         // Create a socket connection with client
45:                         ServerSocketChannel selChannel =
46:                             (ServerSocketChannel) sKey.channel();
47:                         ServerSocket sSock = selChannel.socket();
48:                         Socket connection = sSock.accept();
49:
50:                         // Handle the Finger request
51:                         handleRequest(connection);
52:                         connection.close();
53:                     }
54:                 }
55:             }
56:         } catch (IOException ioe) {
57:             System.out.println(ioe.getMessage());
```

```
58:             }
59:         }
60:
61:     private void handleRequest(Socket connection)
62:         throws IOException {
63:
64:         // Set up input and output
65:         InputStreamReader isr = new InputStreamReader (
66:             connection.getInputStream());
67:         BufferedReader is = new BufferedReader(isr);
68:         PrintWriter pw = new PrintWriter(new
69:             BufferedOutputStream(connection.getOutputStream()),
70:             false);
71:
72:         // Output server greeting
73:         pw.println("Nio Finger Server");
74:         pw.flush();
74:
75:         // Handle user input
76:         String outLine = null;
77:         String inLine = is.readLine();
78:
79:         if (inLine.length() > 0) {
80:             outLine = inLine;
81:         }
82:         readPlan(outLine, pw);
83:
84:         // Clean up
```

```
 85:             pw.flush();
 86:             pw.close();
 87:             is.close();
 88:         }
 89:
 90:         private void readPlan(String userName, PrintWriter pw) {
 91:             try {
 92:                 FileReader file = new FileReader(userName + ".plan");
 93:                 BufferedReader buff = new BufferedReader(file);
 94:                 boolean eof = false;
 95:
 96:                 pw.println("\nUser name: " + userName + "\n");
 97:
 98:                 while (!eof) {
 99:                     String line = buff.readLine();
100:
101:                     if (line == null) {
102:                         eof = true;
103:                     } else {
104:                        pw.println(line);
105:                     }
```

```
106:                 }
107:
108:                 buff.close();
109:             } catch (IOException e) {
110:                 pw.println("User " + userName + " not found.");
111:             }
112:         }
113:
114:         public static void main(String[] arguments) {
115:             FingerServer nio = new FingerServer();
116:         }
117: }
```

```
import java.nio.*;

public class ReadTemps {
    public ReadTemps() {
        int[] temperatures = { 78, 80, 75, 70, 79, 85, 92, 99, 90 };
        IntBuffer tempBuffer = IntBuffer.wrap(temperatures);
        int[] moreTemperatures = { 65, 44, 71 };
        tempBuffer.put(moreTemperatures);
        System.out.println("First int: " + tempBuffer.get());
    }
}
```

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

```
jdbc:derby://localhost:1527/sample
```

```
Connection payday = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/payroll",
    "doc", "1rover1");
```

```
Statement lookSee = payday.createStatement();
```

```
select NAME, CITY from APP.CUSTOMER where (STATE = 'FL')
    order by CITY;
```

```
ResultSet set = looksee.executeQuery(
    "select NAME, CITY from APP.CUSTOMER "
    + " where (STATE = 'FL') order by CITY";
);
```

```
 1: package com.java21days;
 2:
 3: import java.sql.*;
 4:
 5: public class CustomerReporter {
 6:     public static void main(String[] arguments) {
 7:         String data = "jdbc:derby://localhost:1527/sample";
 8:         try (
 9:             Connection conn = DriverManager.getConnection(
10:                 data, "app", "APP");
11:             Statement st = conn.createStatement()) {
12:
13:             Class.forName("org.apache.derby.jdbc.ClientDriver");
14:
15:              ResultSet rec = st.executeQuery(
16:                  "select CUSTOMER_ID, NAME, CITY, STATE " +
17:                  "from APP.CUSTOMER " +
18:                  "order by CUSTOMER_ID");
19:              while (rec.next()) {
```

```
20:                     System.out.println("CUSTOMER_ID:\t"
21:                         + rec.getString(1));
22:                     System.out.println("NAME:\t" + rec.getString(2));
23:                     System.out.println("CITY:\t" + rec.getString(3));
24:                     System.out.println("STATE:\t" + rec.getString(4));
25:                     System.out.println();
26:                 }
27:                 st.close();
28:             } catch (SQLException s) {
29:                 System.out.println("SQL Error: " + s.toString() + " "
30:                     + s.getErrorCode() + " " + s.getSQLState());
31:             } catch (Exception e) {
32:                 System.out.println("Error: " + e.toString()
33:                     + e.getMessage());
34:             }
35:         }
36: }
```

```
select CUSTOMER_ID, NAME, CITY, STATE from APP.CUSTOMER
    order by CUSTOMER_ID;
```

```
PreparedStatement ps = cc.prepareStatement(
    "select * from APP.CUSTOMER where (ZIP=?) "
    + "order by NAME");
```

```
PreparedStatement ps = cc.prepareStatement(
    "insert into APP.CUSTOMER " +
    "VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
```

http://quote.yahoo.com/q?s=fb&d=v1

http://download.finance.yahoo.com/d/quotes.csv?s=FB&f=sl1d1t1c1ohgv&e=.csv

"FB",92.77,"9/25/2015","4:00pm",-1.64,95.85,95.85,92.06,28961622

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.sql.*;
 6: import java.util.*;
 7:
 8: public class QuoteData {
 9:     private String ticker;
10:
11:     public QuoteData(String inTicker) {
12:         ticker = inTicker;
13:     }
14:
15:     private String retrieveQuote() {
16:         StringBuilder builder = new StringBuilder();
17:         try {
18:             URL page = new URL(
19:                 "http://quote.yahoo.com/d/quotes.csv?s=" +
20:                 ticker + "&f=sl1d1t1c1ohgv&e=.csv");
21:             String line;
22:             URLConnection conn = page.openConnection();
23:             conn.connect();
24:             InputStreamReader in = new InputStreamReader(
25:                 conn.getInputStream());
26:             BufferedReader data = new BufferedReader(in);
27:             while ((line = data.readLine()) != null) {
28:                 builder.append(line);
29:                 builder.append("\n");
30:             }
31:         } catch (MalformedURLException mue) {
32:             System.out.println("Bad URL: " + mue.getMessage());
33:         } catch (IOException ioe) {
34:             System.out.println("IO Error:" + ioe.getMessage());
35:         }
```

```
36:         return builder.toString();
37:     }
38:
39:     private void storeQuote(String data) {
40:         StringTokenizer tokens = new StringTokenizer(data, ",");
41:         String[] fields = new String[9];
42:         for (int i = 0; i < fields.length; i++) {
43:             fields[i] = stripQuotes(tokens.nextToken());
44:         }
45:         String datasource = "jdbc:derby://localhost:1527/sample";
46:         try (
47:             Connection conn = DriverManager.getConnection(
48:                 datasource, "app", "app")
49:             ) {
50:
51:             Class.forName("org.apache.derby.jdbc.ClientDriver");
52:             PreparedStatement prep2 = conn.prepareStatement(
53:                 "insert into " +
54:                 "APP.STOCKS(TICKER, PRICE, DATE, CHANGE, LOW, " +
55:                 "HIGH, PRICEOPEN, VOLUME) " +
56:                 "values(?, ?, ?, ?, ?, ?, ?, ?)");
57:             prep2.setString(1, fields[0]);
58:             prep2.setString(2, fields[1]);
59:             prep2.setString(3, fields[2]);
60:             prep2.setString(4, fields[4]);
61:             prep2.setString(5, fields[5]);
62:             prep2.setString(6, fields[6]);
63:             prep2.setString(7, fields[7]);
64:             prep2.setString(8, fields[8]);
65:             prep2.executeUpdate();
```

```
66:                 prep2.close();
67:                 conn.close();
68:             } catch (SQLException sqe) {
69:                 System.out.println("SQL Error: " + sqe.getMessage());
70:             } catch (ClassNotFoundException cnfe) {
71:                 System.out.println(cnfe.getMessage());
72:             }
73:         }
74:
75:         private String stripQuotes(String input) {
76:             StringBuilder output = new StringBuilder();
77:             for (int i = 0; i < input.length(); i++) {
78:                 if (input.charAt(i) != '\"') {
79:                     output.append(input.charAt(i));
80:                 }
81:             }
82:             return output.toString();
83:         }
84:
85:         public static void main(String[] arguments) {
86:             if (arguments.length < 1) {
87:                 System.out.println("Usage: java QuoteData ticker");
88:                 System.exit(0);
89:             }
90:             QuoteData qd = new QuoteData(arguments[0]);
91:             String data = qd.retrieveQuote();
92:             qd.storeQuote(data);
93:         }
94: }
```

```
Connection payday = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/sample", "Doc", "1rover1");
Statement lookSee = payday.CreateStatement();
```

```
Statement lookSee = payday.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY,
   ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

```
public class ArrayClass {

    public static ArrayClass newInstance() {
        count++;
        return new ArrayClass();
    }

    public static void main(String arguments[]) {
        new ArrayClass();
    }

    int count = -1;
}
```

```
 1: <?xml version="1.0" encoding="utf-8"?>
 2: <rss version="2.0">
 3:   <channel>
 4:     <title>Workbench</title>
 5:     <link>http://workbench.cadenhead.org/</link>
 6:     <description>Programming, publishing, and popes</description>
 7:     <docs>http://www.rssboard.org/rss-specification</docs>
 8:     <item>
 9:       <title>Programming Confidence Pool for the World Cup</title>
10:       <link>http://workbench.cadenhead.org/news/739</link>
11:       <pubDate>Wed, 11 Jun 2015 11:49:47 -0400</pubDate>
12:       <guid isPermaLink="false">tag:cadenhead.org,2015:w.739</guid>
13:       <enclosure length="2498623" type="audio/mpeg"
14:           url="http://mp3.cadenhead.org/3679.mp3" />
15:     </item>
16:     <item>
17:       <title>Ghost of Computer Author Past</title>
18:       <link>http://workbench.cadenhead.org/news/737</link>
19:       <pubDate>Mon, 24 Mar 2014 17:00:13 -0400</pubDate>
20:       <guid isPermaLink="false">tag:cadenhead.org,2015:w.737</guid>
21:     </item>
22:     <item>
23:       <title>Interview with Zoe Zolbrod</title>
24:       <link>http://workbench.cadenhead.org/news/736</link>
25:       <pubDate>Fri, 21 Mar 2014 11:12:55 -0400</pubDate>
26:       <guid isPermaLink="false">tag:cadenhead.org,2015:w.736</guid>
27:     </item>
28:   </channel>
29: </rss>
```

```
<!DOCTYPE Library SYSTEM "libraml.dtd">
```

```
1: <?xml version="1.0"?>
2: <rss version="2.0">
3:   <channel>
4:     <title>Workbench</title>
5:     <link>http://workbench.cadenhead.org/</link>
6:   </channel>
7: </rss>
```

```
Element rss = new Element("rss");
```

```
Attribute version = new Attribute("version", "2.0");
```

```
Text titleText = new Text("Workbench");
```

```
Document doc = new Document(rss);
```

```
Element channel = new Element("channel");
Element link = new Element("link");
Text linkText = new Text("http://workbench.cadenhead.org/");
link.appendChild(linkText);
channel.appendChild(link);
```

```
<channel>
    <link>http://workbench.cadenhead.org/</link>
</channel>
```

```
link.appendChild("http://workbench.cadenhead.org/");
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import nu.xom.*;
 5:
 6: public class RssStarter {
 7:     public static void main(String[] arguments) {
 8:         // create an <rss> element to serve as the document's root
 9:         Element rss = new Element("rss");
10:
11:         // add a version attribute to the element
12:         Attribute version = new Attribute("version", "2.0");
13:         rss.addAttribute(version);
14:         // create a <channel> element and make it a child of <rss>
15:         Element channel = new Element("channel");
16:         rss.appendChild(channel);
17:         // create the channel's <title>
18:         Element title = new Element("title");
19:         Text titleText = new Text("Workbench");
20:         title.appendChild(titleText);
21:         channel.appendChild(title);
```

```
22:          // create the channel's <link>
23:          Element link = new Element("link");
24:          Text lText = new Text("http://workbench.cadenhead.org/");
25:          link.appendChild(lText);
26:          channel.appendChild(link);
27:
28:          // create a new document with <rss> as the root element
29:          Document doc = new Document(rss);
30:
31:          // Save the XML document
32:          try (
33:              FileWriter fw = new FileWriter("feed.rss");
34:              BufferedWriter out = new BufferedWriter(fw);
35:          ) {
36:              out.write(doc.toXML());
37:          } catch (IOException ioe) {
38:              System.out.println(ioe.getMessage());
39:          }
40:          System.out.println(doc.toXML());
41:      }
42: }
```

```
<item>
  <title>Free the Bound Periodicals</title>
</item>
```

```
Builder builder = new Builder();
File xmlFile = new File("feed.rss");
Document doc = builder.build(xmlFile);
```

```
Element root = doc.getRootElement();
```

```
Element channel = root.getFirstChildElement("channel");
```

```
Elements children = channel.getChildElements();
```

```
for (int i = 0; i < children.size(); i++) {
    Element link = children.get(i);
}
```

```
Text linkText = (Text) link.getChild(0);
```

```
if (linkText.getValue().equals("http://workbench.cadenhead.org/"))
    // ...
}
```

```
Element channel = domain.getFirstChildElement("channel");
Element link = dns.getFirstChildElement("link");
link.removeChild(0);
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import nu.xom.*;
 5:
 6: public class DomainEditor {
 7:     public static void main(String[] args) throws IOException {
 8:         try {
 9:             // create a tree from the XML document feed.rss
10:             Builder builder = new Builder();
11:             File xmlFile = new File("feed.rss");
12:             Document doc = builder.build(xmlFile);
13:
14:             // get the root element <rss>
15:             Element root = doc.getRootElement();
16:
17:             // get its <channel> element
18:             Element channel = root.getFirstChildElement("channel");
19:
20:             // get its <link> elements
21:             Elements children = channel.getChildElements();
22:             for (int i = 0; i < children.size(); i++) {
23:
24:                 // get a <link> element
25:                 Element link = children.get(i);
26:
27:                 // get its text
28:                 Text linkText = (Text) link.getChild(0);
29:
30:                 // update any link matching a URL
31:                 if (linkText.getValue().equals(
32:                     "http://workbench.cadenhead.org/")) {
33:
34:                     // update the link's text
```

```
35:                         link.removeChild(0);
36:                         link.appendChild("http://www.cadenhead.org/");
37:                     }
38:                 }
39:
40:                 // create new elements and attributes to add
41:                 Element item = new Element("item");
42:                 Element itemTitle = new Element("title");
43:
44:                 // add them to the <channel> element
45:                 itemTitle.appendChild(
46:                     "Free the Bound Periodicals"
47:                 );
48:                 item.appendChild(itemTitle);
49:                 channel.appendChild(item);
50:
51:                 // Save the XML document
52:                 try (
53:                     FileWriter fw = new FileWriter("feed2.rss");
54:                     BufferedWriter out = new BufferedWriter(fw);
55:                 ) {
56:                     out.write(doc.toXML());
57:                 } catch (IOException ioe) {
58:                     System.out.println(ioe.getMessage());
59:                 }
60:                 System.out.println(doc.toXML());
61:             } catch (ParsingException pe) {
62:                 System.out.println("Parse error: " + pe.getMessage());
63:                 pe.printStackTrace();
64:                 System.exit(-1);
65:             }
66:         }
67: }
```

```
 1: <?xml version="1.0" encoding="ISO-8859-1"?>
 2: <!--File created Sat Sep 26 23:17:49 EDT 2015-->
 3: <rss version="2.0">
 4:   <channel>
 5:     <title>Workbench</title>
 6:     <link>http://www.cadenhead.org/</link>
 7:     <item>
 8:       <title>Free the Bound Periodicals</title>
 9:     </item>
10:   </channel>
11: </rss>
```

```
FileOutputStream fos = new FileOutputStream("feed3.rss");
Serializer output = new Serializer(fos, "ISO-8859-1");
```

```
Builder builder = new Builder();
Document doc = builder.build(arguments[0]);
Comment timestamp = new Comment("File created " +
    new java.util.Date());
doc.insertChild(timestamp, 0);
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import nu.xom.*;
 5:
 6: public class DomainWriter {
 7:     public static void main(String[] args) throws IOException {
 8:         try {
 9:             // Create a tree from an XML document
10:             // specified as a command-line argument
11:             Builder builder = new Builder();
12:             File xmlFile = new File("feed2.rss");
13:             Document doc = builder.build(xmlFile);
14:
15:             // Create a comment with the current time and date
16:             Comment timestamp = new Comment("File created "
17:                 + new java.util.Date());
18:
19:             // Add the comment above everything else in the
20:             // document
```

```
21:             doc.insertChild(timestamp, 0);
22:
23:             // Create a file output stream to a new file
24:             FileOutputStream f = new FileOutputStream("feed3.rss");
25:
26:             // Using a serializer with indention set to 2 spaces,
27:             // write the XML document to the file
28:             Serializer output = new Serializer(f, "ISO-8859-1");
29:             output.setIndent(2);
30:             output.write(doc);
31:         } catch (ParsingException pe) {
32:             System.out.println("Parsing error: " + pe.getMessage());
33:             pe.printStackTrace();
34:             System.exit(-1);
35:         }
36:     }
37: }
```

```
 1: package com.java21days;
 2:
 3: import nu.xom.*;
 4:
 5: public class RssFilter {
 6:     public static void main(String[] arguments) {
 7:
 8:         if (arguments.length < 2) {
 9:             System.out.println("Usage: java RssFilter file term");
10:             System.exit(-1);
11:         }
12:
13:         // Save the RSS location and search term
14:         String rssFile = arguments[0];
15:         String term = arguments[1];
16:
17:         try {
18:             // Fill a tree with an RSS file's XML data
19:             // The file can be local or something on the
20:             // Web accessible via a URL.
21:             Builder bob = new Builder();
22:             Document doc = bob.build(rssFile);
23:
24:             // Get the file's root element (<rss>)
25:             Element rss = doc.getRootElement();
26:
27:             // Get the element's version attribute
28:             Attribute rssVersion = rss.getAttribute("version");
29:             String version = rssVersion.getValue();
30:
31:             // Add the DTD for RSS 0.91 feeds, if needed
32:             if ( (version.equals("0.91")) &
33:                  (doc.getDocType() == null) ) {
34:
```

```
35:                 DocType rssDtd = new DocType("rss",
36:                 "http://my.netscape.com/publish/formats/rss-0.91.dtd");
37:                 doc.insertChild(rssDtd, 0);
38:             }
39:
40:             // Get the first (and only) <channel> element
41:             Element channel = rss.getFirstChildElement("channel");
42:
43:             // Get its <title> element
44:             Element title = channel.getFirstChildElement("title");
45:             Text titleText = (Text) title.getChild(0);
46:
47:             // Change the title to reflect the search term
48:             titleText.setValue(titleText.getValue() +
49:                 ": Search for " + term + " articles");
50:
51:             // Get all of the <item> elements and loop through them
52:             Elements items = channel.getChildElements("item");
53:             for (int i = 0; i < items.size(); i++) {
54:                 // Get an <item> element
55:                 Element item = items.get(i);
56:
```

```
57:                     // Look for a <title> element inside it
58:                     Element iTitle = item.getFirstChildElement("title");
59:
60:                     // If found, look for its contents
61:                     if (iTitle != null) {
62:                         Text iTitleText = (Text) iTitle.getChild(0);
63:
64:                         // If the search text is not found in the item,
65:                         // delete it from the tree
66:                         if (iTitleText.toString().indexOf(term) == -1) {
67:                             channel.removeChild(item);
68:                         }
69:                     }
70:                 }
71:
72:                 // Display the results with a serializer
73:                 Serializer output = new Serializer(System.out);
74:                 output.setIndent(2);
75:                 output.write(doc);
76:             } catch (Exception exc) {
77:                 System.out.println("Error: " + exc.getMessage());
78:                 exc.printStackTrace();
79:             }
80:         }
81: }
```

```
public class NameDirectory {
    String[] names;
    int nameCount;

    public NameDirectory() {
        names = new String[20];
        nameCount = 0;
    }

    public void addName(String newName) {
        if (nameCount < 20) {
            // answer goes here
        }
    }
}
```

```
 1: POST /XMLRPC HTTP/1.0
 2: Host: www.advogato.org
 3: Connection: Close
 4: Content-Type: text/xml
 5: Content-Length: 151
 6: User-Agent: OSE/XML-RPC
 7:
 8: <?xml version="1.0"?>
 9: <methodCall>
10:    <methodName>test.square</methodName>
11:    <params>
12:       <param>
13:          <value>
14:             <int>13</int>
15:          </value>
16:       </param>
17:    </params>
18: </methodCall>
```

```
 1: HTTP/1.0 200 OK
 2: Date: Sat, 27 Sep 2015 04:44:13 GMT
 3: Server: Apache/2.2.3·(CentOS)
 4: ETag: "PbT9cMgXsXnw52OqREFNAA=="
 5: Content-MD5: PbT9cMgXsXnw52OqREFNAA==
 6: Content-Length: 157
 7: Connection: close
 8: Content-Type: text/xml
 9:
10: <?xml version="1.0"?>
11: <methodResponse>
12:   <params>
13:     <param>
14:       <value>
15:         <int>169</int>
16:       </value>
17:     </param>
18:   </params>
19: </methodResponse>
```

```
XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
URL server = new URL("http://cadenhead.org:4413/");
config.setServerURL(server);
XmlRpcClient client = new XmlRpcClient();
client.setConfig(config);
```

```
String code = "conical";
Double xValue = new Double(175);
ArrayList parameters = new ArrayList();
parameters.add(code);
parameters.add(xValue);
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import java.net.*;
 5: import java.util.*;
 6: import org.apache.xmlrpc.*;
 7: import org.apache.xmlrpc.client.*;
 8:
 9: public class SiteClient {
10:     public static void main(String arguments[]) {
11:         SiteClient client = new SiteClient();
12:         try {
13:             HashMap<String, String> resp = client.getRandomSite();
14:             // Report the results
15:             if (resp.size() > 0) {
16:                 System.out.println("URL: " + resp.get("url")
17:                     + "\nTitle: " + resp.get("title")
18:                     + "\nDescription: " + resp.get("description"));
19:             }
20:         } catch (IOException ioe) {
21:             System.out.println("Exception: " + ioe.getMessage());
22:         } catch (XmlRpcException xre) {
```

```
23:             System.out.println("Exception: " + xre.getMessage());
24:         }
25:     }
26:
27:     public HashMap getRandomSite()
28:         throws IOException, XmlRpcException {
29:
30:             // Create the client
31:             XmlRpcClientConfigImpl config = new
32:                 XmlRpcClientConfigImpl();
33:             URL server = new URL("http://localhost:4413/");
34:             config.setServerURL(server);
35:             XmlRpcClient client = new XmlRpcClient();
36:             client.setConfig(config);
37:             // Create the parameters for the request
38:             ArrayList params = new ArrayList();
39:             // Send the request and get the response
40:             HashMap result = (HashMap) client.execute(
41:                 "dmoz.getRandomSite", params);
42:             return result;
43:     }
44: }
```

```
WebServer server = new WebServer(4413);
XmlRpcServer xmlRpcServer = server.getXmlRpcServer();
```

```
PropertyHandlerMapping phm = new PropertyHandlerMapping();
```

```
phm.addHandler("dmoz", DmozHandlerImpl.class);
xmlRpcServer.setHandlerMapping(phm);
```

```
 1: package com.java21days;
 2:
 3: import java.io.*;
 4: import org.apache.xmlrpc.*;
 5: import org.apache.xmlrpc.server.*;
 6: import org.apache.xmlrpc.webserver.*;
 7:
 8: public class DmozServer {
 9:     public static void main(String[] arguments) {
10:         try {
11:             startServer();
12:         } catch (IOException ioe) {
13:             System.out.println("Server error: " +
14:                 ioe.getMessage());
15:         } catch (XmlRpcException xre) {
16:             System.out.println("XML-RPC error: " +
17:                 xre.getMessage());
18:         }
19:     }
20:
21:     public static void startServer() throws IOException,
22:         XmlRpcException {
```

```
23:
24:         // Create the server
25:         System.out.println("Starting Dmoz server ...");
26:         WebServer server = new WebServer(4413);
27:         XmlRpcServer xmlRpcServer = server.getXmlRpcServer();
28:         PropertyHandlerMapping phm = new PropertyHandlerMapping();
29:
30:         // Register the handler
31:         phm.addHandler("dmoz", DmozHandlerImpl.class);
32:         xmlRpcServer.setHandlerMapping(phm);
33:
34:         // Start the server
35:         server.start();
36:         System.out.println("Accepting requests ...");
37:     }
38: }
```

```
1: package com.java21days;
2:
3: import java.util.*;
4:
5: public interface DmozHandler {
6:     public HashMap getRandomSite();
7: }
```

```
 1: package com.java21days;
 2:
 3: import java.sql.*;
 4: import java.util.*;
 5:
 6: public class DmozHandlerImpl implements  DmozHandler {
 7:
 8:     public HashMap getRandomSite() {
 9:         Connection conn = getMySqlConnection();
10:         HashMap<String, String> response = new HashMap<>();
11:         try {
12:             Statement st = conn.createStatement();
13:             ResultSet rec = st.executeQuery(
14:                 "SELECT * FROM cooldata ORDER BY RAND() LIMIT 1");
15:             if (rec.next()) {
16:                 response.put("url", rec.getString("url"));
17:                 response.put("title", rec.getString("title"));
18:                 response.put("description",
19:                     rec.getString("description"));
20:             } else {
21:                 response.put("error", "no database record found");
22:             }
```

```
23:             st.close();
24:             rec.close();
25:             conn.close();
26:         } catch (SQLException sqe) {
27:             response.put("error", sqe.getMessage());
28:         }
29:         return response;
30:     }
31:
32:     private Connection getMySqlConnection() {
33:         Connection conn = null;
34:         String data = "jdbc:mysql://localhost/cool";
35:         try {
36:             Class.forName("com.mysql.jdbc.Driver");
37:             conn = DriverManager.getConnection(
38:                 data, "cool", "mrfreeze");
39:         } catch (SQLException s) {
40:             System.out.println("SQL Error: " + s.toString() + " "
41:                 + s.getErrorCode() + " " + s.getSQLState());
42:         } catch (Exception e) {
43:             System.out.println("Error: " + e.toString()
44:                 + e.getMessage());
45:         }
46:         return conn;
47:     }
48: }
```

```
byte[] data = (byte[]) fromServer;
```

```
public class Operation {
    public static void main(String[] arguments) {
        int x = 1;
        int y = 3;
        if ((x != 1) && (y++ == 3)) {
            y = y + 2;
        }
    }
}
```

```
ERROR: x86 emulation currently requires hardware acceleration!
Please ensure Intel HAXM is properly installed and usable. CPU
acceleration status: HAX kernel module is not installed!
```

```
 1: package com.java21days.santa;
 2:
 3: import android.support.v7.app.AppCompatActivity;
 4: import android.os.Bundle;
 5: import android.view.Menu;
 6: import android.view.MenuItem;
 7:
 8: public class SantaActivity extends AppCompatActivity {
 9:
10:     @Override
11:     protected void onCreate(Bundle savedInstanceState) {
12:         super.onCreate(savedInstanceState);
13:         setContentView(R.layout.activity_santa);
14:     }
15:
16:     @Override
17:     public boolean onCreateOptionsMenu(Menu menu) {
18:         // Inflate the menu; this adds items to the action bar
19:         getMenuInflater().inflate(R.menu.menu_santa, menu);
20:         return true;
21:     }
```

```
22:
23:     @Override
24:     public boolean onOptionsItemSelected(MenuItem item) {
25:         // Handle action bar item clicks here. The action bar will
26:         // automatically handle clicks on Home/Up button, so long
27:         // as you specify a parent activity in AndroidManifest.xml.
28:         int id = item.getItemId();
29:
30:         // noinspection SimplifiableIfStatement
31:         if (id == R.id.action_settings) {
32:             return true;
33:         }
34:
35:         return super.onOptionsItemSelected(item);
36:     }
37: }
```

```
public void processClicks(View display) {
    Intent action = null;
    int id = display.getId();
}
```

```
action = new Intent(Intent.ACTION_DIAL, Uri.parse(
    "tel:877-446-6723"));
action = new Intent(Intent.ACTION_VIEW, Uri.parse(
    "http://www.noradsanta.org"));
action = new Intent(Intent.ACTION_VIEW, Uri.parse(
    "geo:0,0?q=101 Saint Nicholas Dr., North Pole, AK"));
```

```
 1: package com.java21days.santa;
 2:
 3: import android.content.Intent;
 4: import android.net.Uri;
 5: import android.support.v7.app.AppCompatActivity;
 6: import android.os.Bundle;
 7: import android.view.Menu;
 8: import android.view.MenuItem;
 9: import android.view.View;
10:
11: public class SantaActivity extends AppCompatActivity {
12:
13:     @Override
14:     protected void onCreate(Bundle savedInstanceState) {
15:         super.onCreate(savedInstanceState);
16:         setContentView(R.layout.activity_santa);
17:     }
18:
19:     public void processClicks(View display) {
20:         Intent action = null;
21:         int id = display.getId();
22:
23:         switch (id) {
24:             case (R.id.imageButton):
25:                 action = new Intent(Intent.ACTION_DIAL,
26:                     Uri.parse("tel:877-446-6723"));
27:                 break;
28:             case (R.id.imageButton2):
29:                 action = new Intent(Intent.ACTION_VIEW,
30:                     Uri.parse("http://www.noradsanta.org"));
31:                 break;
32:             case (R.id.imageButton3):
```

```
33:                     action = new Intent(Intent.ACTION_VIEW,
34:                         Uri.parse("geo:0,0?q=101 Saint Nicholas Dr., North
➥Pole, AK"));
35:                         break;
36:                 default:
37:                     break;
38:                 }
39:             startActivity(action);
40:         }
41:
42:         @Override
43:         public boolean onCreateOptionsMenu(Menu menu) {
44:             // Inflate the menu; this adds items to the action bar
45:             getMenuInflater().inflate(R.menu.menu_santa, menu);
46:             return true;
47:         }
48:
49:         @Override
50:         public boolean onOptionsItemSelected(MenuItem item) {
51:             // Handle action bar item clicks here. The action bar will
52:             // automatically handle clicks on Home/Up button, so long
53:             // as you specify a parent activity in AndroidManifest.xml.
54:             int id = item.getItemId();
55:
56:             //noinspection SimplifiableIfStatement
57:             if (id == R.id.action_settings) {
58:                 return true;
59:             }
60:
61:             return super.onOptionsItemSelected(item);
62:         }
63: }
```

```
public class CharCase {
    public static void main(String[] arguments) {
        float x = 9;
        float y = 5;
        char c = '1';
        switch (c) {
            case 1:
                x = x + 2;
            case 2:
                x = x + 3;
            default:
                x = x + 1;
        }
        System.out.println("Value of x: " + x);
    }
}
```

```
System.out.println("I am Spartacus!");
```

```
 1: /*
 2:  * To change this template, choose Tools | Templates
 3:  * and open the template in the editor.
 4:  */
 5:
 6: /**
 7:  *
 8:  * @author User
 9:  */
10: public class Spartacus {
11:
12:     /**
13:      * @param args the command line arguments
14:      */
15:     public static void main(String[] args) {
16:         // TODO code application logic here
17:         System.out.println("I am Spartacus!");
18:
19:     }
20:
21: }
```

```
ERROR: x86 emulation currently requires hardware acceleration!
Please ensure Intel HAXM is properly installed and usable. CPU
acceleration status: HAX kernel module is not installed!
```

```
;c:\Program Files\Java\jdk1.8.0_60\bin
```

```
PATH c:\"Program Files"\Java\jdk1.8.0_60\bin;%PATH%
```

```
Palindrome.java:2: Class Font not found in type declaration.
```

```
1: public class HelloUser {
2:     public static void main(String[] arguments) {
3:         String username = System.getProperty("user.name");
4:         System.out.println("Hello " + username);
5:     }
6: }
```

```
Exception in thread "main"
java.lang.NoClassDefFoundError: HelloUser/class
```

```
.;C:\"Program Files"\Java\jdk1.8.0_60\lib\tools.jar
```

```
SET CLASSPATH=%CLASSPATH%;.;rightlocation
```

```
SET CLASSPATH=%CLASSPATH%;.;c:\"Program Files"\Java\jdk1.8.0_60\lib\tools.jar
```

```
java VideoBook add DVD "Broadcast News"
```

```
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

```
javac -deprecation OldVideoBook.java
```

```
public static void main(String[] arguments) {
    // method here
}
```

```
java org.cadenhead.auction.SellItem
```

```
java -cp . org.cadenhead.auction.SellItem
```

```
javac BuyItem.java SellItem.java
```

```
javac -deprecation SellItem.java
```

```
appletviewer http://www.javaonthebrain.com
```

```
 1: import java.awt.*;
 2:
 3: public class AppInfo extends javax.swing.JApplet {
 4:     String name, date;
 5:     int version;
 6:
 7:     public String getAppletInfo() {
 8:         String response = "This applet demonstrates the "
 9:             + "use of the Applet's Info feature.";
10:         return response;
11:     }
12:
13:     public String[][] getParameterInfo() {
14:         String[] p1 = { "Name", "String", "Programmer's name" };
15:         String[] p2 = { "Date", "String", "Today's date" };
16:         String[] p3 = { "Version", "int", "Version number" };
17:         String[][] response = { p1, p2, p3 };
18:         return response;
19:     }
20:
21:     public void init() {
22:         name = getParameter("Name");
23:         date = getParameter("Date");
```

```
24:         String versText = getParameter("Version");
25:         if (versText != null) {
26:             version = Integer.parseInt(versText);
27:         }
28:     }
29:
30:     public void paint(Graphics screen) {
31:         Graphics2D screen2D = (Graphics2D) screen;
32:         screen2D.drawString("Name: " + name, 5, 50);
33:         screen2D.drawString("Date: " + date, 5, 100);
34:         screen2D.drawString("Version: " + version, 5, 150);
35:     }
36: }
```

This applet demonstrates the use of the Applet's Info feature.

```
1: <applet code="AppInfo.class" height="200" width="170">
2: <param name="Name" value="Rogers Cadenhead">
3: <param name="Date" value="08/15/15">
4: <param name="Version" value="7">
5: </applet>
```

```
/** A descriptive sentence or paragraph.
  * @tag1 Description of this tag.
  * @tag2 Description of this tag.
 */
```

```
 1: import java.awt.*;
 2:
 3: /** This class displays the values of three parameters:
 4:   * Name, Date and Version.
 5:   * @author <a href="http://java21days.com/">Rogers Cadenhead</a>
 6:   * @version 7.0
 7:  */
 8: public class AppInfo2 extends javax.swing.JApplet {
 9:     /**
10:       * @serial The programmer's name.
11:      */
12:     String name;
13:     /**
14:       * @serial The current date.
15:      */
16:     String date;
17:     /**
18:       * @serial The program's version number.
19:      */
20:     int version;
21:
22:     /**
23:       * This method describes the applet for any browsing tool that
24:       * requests information from the program.
25:       * @return A String describing the applet.
26:      */
27:     public String getAppletInfo() {
28:         String response = "This applet demonstrates the "
29:             + "use of the Applet's Info feature.";
```

```
30:         return response;
31:     }
32:
33:     /**
34:       * This method describes the parameters that the applet can take
35:       * for any browsing tool that requests this information.
36:       * @return An array of String[] objects for each parameter.
37:      */
38:     public String[][] getParameterInfo() {
39:         String[] p1 = { "Name", "String", "Programmer's name" };
40:         String[] p2 = { "Date", "String", "Today's date" };
41:         String[] p3 = { "Version", "int", "Version number" };
42:         String[][] response = { p1, p2, p3 };
43:         return response;
44:     }
45:
46:     /**
47:       * This method is called when the applet is first initialized.
48:      */
49:     public void init() {
50:         name = getParameter("Name");
51:         date = getParameter("Date");
52:         String versText = getParameter("Version");
53:         if (versText != null) {
54:             version = Integer.parseInt(versText);
55:         }
56:     }
```

```
57:
58:     /**
59:       * This method is called when the applet's display window is
60:       * being repainted.
61:      */
62:     public void paint(Graphics screen) {
63:         Graphics2D screen2D = (Graphics2D) screen;
64:         screen.drawString("Name: " + name, 5, 50);
65:         screen.drawString("Date: " + date, 5, 100);
66:         screen.drawString("Version: " + version, 5, 150);
67:     }
68: }
```

```
javadoc -author -version AppInfo2.java
```

```
javadoc -author -version -d C:\JavaDocs\ AppInfo2.java
```

```
jar cf Animate.jar *.class *.gif
```

```
jar cf MusicLoop.jar MusicLoop.class muskratLove.mp3 shopAround.mp3
```

```
<applet code="MusicLoop.class" archive="MusicLoop.jar" width="45"
➥height="42">
</applet>
```

```
<object code="MusicLoop.class" width="45" height="42">
    <param name="archive" value="MusicLoop.jar">
</object>
```

```
Deferring breakpoint WriteBytes:14
It will be set after the class is loaded.
```

```
run WriteBytes
VM Started: Set deferred breakpoint WriteBytes:14

Breakpoint hit: "thread=main", WriteBytes.main(), line=14 bci=413
14                  for (int i = 0; i < data.length; i++)
```

```
appletviewer -debug AppInfo.html
```

```
stop in AppInfo.getAppletInfo
```

```
java -Duser.timezone=Asia/Jakarta Auctioneer
```

```
String[] ids = java.util.TimeZone.getAvailableIDs();
for (int i = 0; i < ids.length; i++) {
    System.out.println(ids[i]);
}
```

```
1: class ItemProp {
2:     public static void main(String[] arguments) {
3:         String n = System.getProperty("item.name");
4:         System.out.println("The item is named " + n);
5:     }
6: }
```

```
java -Ditem.name="Microsoft Bob" ItemProp
```

The item is named Microsoft Bob

```
appletviewer -J-Dtimezone=Asia/Jakarta AppInfo.html
```

```
keytool -genkey -alias examplekey -keypass swordfish
```

```
jarsigner -storepass bazinga -keypass swordfish Animate.jar examplekey
```