GLOBAL EDITION

Android™

How to Program

SECOND EDITION

Paul Deitel • Harvey Deitel • Abbey Deitel

ALWAYS LEARNING

PEARSON

ONLINE ACCESS

Thank you for purchasing a new copy of ***Android*™ *How to Program,*** **Second Edition**. Your textbook includes 12 months of prepaid access to the book's Companion Website. This prepaid subscription provides you with full access to the following student support areas:

- Source code
- Premium web chapters

Use a coin to scratch off the coating and reveal your student access code.
Do not use a knife or other sharp object as it may damage the code.

To access the ***Android How to Program,*** **Second Edition**, Companion Website for the first time, you will need to register online using a computer with an Internet connection and a web browser. The process takes just a couple of minutes and only needs to be completed once.

1. Go to **www.pearsonglobaleditions.com/deitel**
2. Click on **Companion Website**.
3. Click on the **Register** button.
4. On the registration page, enter your student access code* found beneath the scratch-off panel. Do not type the dashes. You can use lower- or uppercase.
5. Follow the on-screen instructions. If you need help at any time during the online registration process, simply click the **Need Help?** icon.
6. Once your personal Login Name and Password are confirmed, you can begin using the *Android How to Program* Companion Website!

To log in after you have registered:

You only need to register for this Companion Website once. After that, you can log in any time at **www.pearsonglobaleditions.com/deitel** by providing your Login Name and Password when prompted.

*Important: The access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable. If this access code has already been revealed, it may no longer be valid. If this is the case, you can purchase a subscription by going to **www.pearsonglobaleditions.com/deitel** going to the Android book and following the on-screen instructions.

Android™ HOW TO PROGRAM

SECOND EDITION
Global Edition

Android™ HOW TO PROGRAM

SECOND EDITION
Global Edition

Paul Deitel • Harvey Deitel • Abbey Deitel
Deitel & Associates, Inc.
Global Edition contributions by Muthuraj M.

DEITEL®

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director, ECS: ***Marcia Horton***
Head of Learning Asset Acquisition, Global Edition: ***Laura Dent***
Executive Editor: ***Tracy Johnson (Dunkelberger)***
Director of Marketing: ***Christy Lesko***
Marketing Manager: ***Yez Alayan***
Marketing Assistant: ***Jon Bryant***
Director of Program Management: ***Erin Gregg***
Program Management-Team Lead: ***Scott Disanno***
Program Manager: ***Carole Snyder***
Project Management-Team Lead: ***Laura Burgess***
Project Manager: ***Robert Engelhardt***
Publishing Administrator and Business Analyst, Global Edition: ***Shokhi Shah Khandelwal***
Acquisitions Editor, Global Edition: ***Karthik Subramanian***
Assistant Project Editor, Global Edition: ***Sinjita Basu***
Media Producer, Global Edition: ***M. Vikram Kumar***
Senior Manufacturing Controller, Production, Global Edition: ***Trudy Kimber***
Procurement Specialist: ***Linda Sager***
Permissions Supervisor: ***Michael Joyce***
Permissions Administrator: ***Jenell Forschler***
Director, Image Asset Services: ***Annie Atherton***
Manager, Visual Research: ***Karen Sanatar***
Media Project Manager: ***Renata Butera***
Cover Designer: ***Shree Inbakumar***
Cover Photo: ***Kirill__M/ Shutterstock***
Cover Printer: ***Courier Westford***

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

ISBN 10: 0-273-79339-X

ISBN 13: 978-0-273-79339-7

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1
14 13 12 11 10

Typeset in Adobe Garamond by GEX Publishing Services.

Printed and bound by Courier Westford in the United States of America.

In Memory of Amar G. Bose, MIT Professor and Founder and Chairman of the Bose Corporation:

It was a privilege being your student—and members of the next generation of Deitels, who heard our dad say how your classes inspired him to do his best work.

You taught us that if we go after the really hard problems, then great things can happen.

Harvey Deitel
Paul and Abbey Deitel

Trademarks

Contents

Preface

Build a better mousetrap, and the world will beat a path to your door.
—Ralph Waldo Emerson

Science and technology and the various forms of art,
all unite humanity in a single and interconnected system.
—Zhores Aleksandrovich Medvede

Welcome to the dynamic world of Android smartphone and tablet app development with the Android Software Development Kit (SDK), the Java™ programming language, the Android Development Tools IDE, and the new and rapidly evolving Android Studio. We present leading-edge mobile computing technologies for students, instructors and professional software developers.

Android How to Program, 2/e

With this unique book—the second edition of the world's first Android computer science textbook—you can learn Android even if you don't know Java and even if you're a programming novice. This book includes a complete, 300-page introduction to the Java core programming concepts that you'll need when developing Android apps. The Java content is appropriate for programming novices.

Android How to Program, 2/e was formed by merging

- our professional book *Android for Programmers: An App-Driven Approach, 2/e, Volume 1*
- additional online chapters selected from *Android for Programmers: An App-Driven Approach, 2/e, Volume 2*
- condensed, introductory core content on object-oriented Java programming from our college textbook *Java How to Program, 9/e*
- hundreds of Android short-answer questions and app-development exercises we created for this book—most are in the book and many of the short-answer questions are in the test-item file for instructors.

We scoured the Android material, especially the fully coded Android apps, and enumerated the Java features that you'll need to build these and similar apps. Then we extracted the corresponding Java content from *Java How to Program, 9/e*. That's a 1500-page book, so it was challenging to whittle down that much content and keep it friendly, even for programming novices.

When you study the Android content, you'll be thinking like a developer from the start. You're going to study and build lots of real stuff and you'll face the kinds of challenges professional developers must deal with. We'll point you to the online documentation and forums where you can find additional information and get answers to your questions. We'll also encourage you to read, modify and enhance open-source code as part of your learning process.

Intended Audiences

There are several audiences for this book. Most commonly, it will be used in upper-level elective college courses and industry professional courses for people familiar with object-oriented programming but who may or may not know Java and want to learn Android app development.

Uniquely, the book can also be used in introductory courses like CS1, intended for programming novices. We recommend that schools typically offering many sections of CS1 in Java consider designating one or two sections for ambitious students who have at least some prior programming experience and who want to work hard to learn a good amount of Java and Android in an aggressively paced one-semester course. The schools may want to list the courses with "honors" or "accelerated" designations. The book works especially well in two-semester introductory programming sequences where the introduction to Java is covered first.

App-Development Courses

In 2007, Stanford offered a new course called Creating Engaging Facebook Apps. Students worked in teams developing apps, some of which landed in Facebook's top 10, earning some of the student developers millions of dollars.[1] This course gained wide recognition for encouraging student creativity and teamwork. Scores of colleges now offer app-development courses across many social networking and mobile platforms such as Android and iOS. We encourage you to read the online mobile app development syllabi and check out the YouTube™ videos created by instructors and students for many of these courses.

Android Ecosystem: Competition, Innovation, Explosive Growth and Opportunities

Sales of Android devices and app downloads have been growing exponentially. The first-generation Android phones were released in October 2008. A study by Strategy Analytics showed that by October 2013, Android had 81.3% of the global smartphone market share, compared to 13.4% for Apple, 4.1% for Microsoft and 1% for Blackberry.[2] According to an IDC report, by the end of the first quarter of 2013 Android had 56.5% of the global tablet market share, compared to 39.6% for Apple's iPad and 3.7% for Microsoft Windows tablets.[3]

There are now over one billion Android smartphones and tablets in use,[4] and more than 1.5 million Android devices are being activated daily.[5] According to IDC, Samsung

1. `http://www.businessinsider.com/these-stanford-students-made-millions-taking-a-class-on-facebook-2011-5`.
2. `http://blogs.strategyanalytics.com/WSS/post/2013/10/31/Android-Captures-Record-81-Percent-Share-of-Global-Smartphone-Shipments-in-Q3-2013.aspx`.
3. `http://www.idc.com/getdoc.jsp?containerId=prUS24093213`.
4. `http://www.android.com/kitkat`.
5. `http://www.technobuffalo.com/2013/04/16/google-daily-android-activations-1-5-million`.

is the leading Android manufacturer, accounting for nearly 40% of Android device shipments in the third quarter of 2013.

Billions of apps have been downloaded from Google Play™—Google's marketplace for Android Apps. The opportunities for Android app developers are enormous.

Fierce competition among popular mobile platforms and carriers is leading to rapid innovation and falling prices. Competition among the dozens of Android device manufacturers is driving hardware and software innovation within the Android community.

App-Driven Approach

At the heart of the book is our *app-driven approach*—we present concepts in the context of *seven complete working Android apps* in the print book and more online. We begin each of the app chapters with an *introduction* to the app, an app *test-drive* showing one or more *sample executions*, and a *technologies overview*. We build the app's GUI and resource files. Then we proceed with a detailed *code walkthrough* of the app's source code in which we discuss the programming concepts and demonstrate the functionality of the Android APIs used in the app. All the source code is available at the book's Companion Website `www.pearsonglobaleditions.com/Deitel`. We recommend that you have the source code open in the IDE as you read the book. Figure 1 lists the book's apps and the key technologies we used to build each.

App	Technologies
Chapter 2, **Welcome** App	The Android Developer Tools (the Eclipse IDE and the ADT Plugin), visual GUI design, layouts, `TextViews`, `ImageViews`, accessibility and internationalization.
Chapter 3, **Tip Calculator** App	`GridLayout`, `LinearLayout`, `EditText`, `SeekBar`, event handling, `NumberFormat` and defining app functionality with Java.
Chapter 4, **Twitter® Searches** App	`SharedPreferences`, collections, `ImageButton`, `ListView`, `ListActivity`, `ArrayAdapter`, implicit intents and `AlertDialogs`.
Chapter 5, **Flag Quiz** App	Fragments, menus, preferences, `AssetManager`, tweened animations, `Handler`, `Toasts`, Explicit `Intents`, layouts for multiple device orientations.
Chapter 6, **Cannon Game** App	Listening for touches, frame-by-frame animation, graphics, sound, threading, `SurfaceView` and `SurfaceHolder`.
Chapter 7, **Doodlz** App	Two-dimensional graphics, Canvas, Bitmap, accelerometer, `SensorManager`, multitouch events, `MediaStore`, printing and Immersive Mode.
Chapter 8, **Address Book** App	`AdapterViews` and `Adapters`

Fig. 1 | *Android How to Program* apps in the print book.

Online Chapters and Book Updates

The Companion Website contains additional app-development chapters that introduce property animation, Google Play game services, video, speech synthesis and recognition, GPS, the Maps API, the compass, object serialization, Internet-enabled apps, audio recording and playback, Bluetooth®, HTML5 mobile apps and more. **Most of these chapters will be available for fall 2014 courses. For the status of the online chapters and for continuing book updates, visit**

```
    www.pearsonglobaleditions.com/Deitel
```

Join the Deitel communities on Facebook® (`http://www.deitel.com/deitelfan`), Twitter® (`@deitel`), LinkedIn® (`http://bit.ly/DeitelLinkedIn`) Google+™ (`http://google.com/+DeitelFan`), and YouTube™ (`http://youtube.com/user/DeitelTV`) and subscribe to the *Deitel® Buzz Online* newsletter (`http://www.deitel.com/newsletter/subscribe.html`).

Copyright Notice and Code License

Getting up to Speed in Java and XML

The Android portion of this book assumes that you already know Java and object-oriented programming. If you're not familiar with these, the appendices provide a condensed, friendly introduction to Java and the object-oriented programming techniques you'll need to develop Android apps. If you're interested in learning Java in more depth, you may want to check out the comprehensive treatment in our textbook *Java How to Program, 10/e* `www.pearsonglobaleditions.com/Deitel`.

Because of the improved Android development tools, we were able to eliminate almost all XML markup in this edition. There are still two small, easy-to-understand XML files you'll need to manipulate. If you're not familiar with XML, see these online tutorials:

- `http://www.deitel.com/articles/xml_tutorials/20060401/XMLBasics/`
- `http://www.deitel.com/articles/xml_tutorials/20060401/XMLStructuringData/`
- `http://www.ibm.com/developerworks/xml/newto/`
- `http://www.w3schools.com/xml/xml_whatis.asp`

Key Features of *Android How to Program, 2/e*

- ***Android SDK 4.3 and 4.4.*** We cover various new Android Software Development Kit (SDK) 4.3 and 4.4 features. [*Note:* The apps in this book are configured to run on Android devices with Android 4.3 and higher; however, most apps will work in 4.0 and higher by changing their minimum required SDK.]
- ***Fragments.*** Starting with Chapter 5, we use `Fragments` to create and manage portions of each app's GUI. You can combine several fragments to create user interfaces that take advantage of tablet screen sizes. You also can easily interchange fragments to make your GUIs more dynamic, as you'll do in Chapter 8.
- ***Support for multiple screen sizes and resolutions.*** Throughout the app chapters we demonstrate how to use Android's mechanisms for automatically choosing resources (layouts, images, etc.) based on a device's size and orientation.
- ***Eclipse-Based Android Development Tools (ADT) coverage in the print book.*** The free Android Development Tools (ADT) integrated development environment (IDE)—which includes Eclipse and the ADT plugin—combined with the free Java Development Kit (JDK) provide all the software you'll need to create, run and debug Android apps, export them for distribution (e.g., upload them to Google Play™) and more.
- ***Android Studio.*** This is the preferred IDE for the future of Android app development. Because this IDE is evolving quickly, we put our discussions of it online at:

```
www.pearsonglobaleditions.com/Deitel
```

- ***Immersive Mode.*** The status bar at the top of the screen and the menu buttons at the bottom can be hidden, allowing your apps to fill more of the screen. Users can access the status bar by swiping down from the top of the screen, and the system bar (with the back button, home button and recent apps button) by swiping up from the bottom.
- ***Printing Framework.*** Android 4.4 KitKat allows you to add printing functionality to your apps, such as locating available printers over Wi-Fi or the cloud, selecting the paper size and specifying which pages to print.
- ***Testing on Android Smartphones, Tablets and the Android Emulator.*** For the best app-development experience, you should test your apps on actual Android smartphones and tablets. You can still have a meaningful experience using the Android emulator (see the Before You Begin section), however it's processor-intensive and can be slow—particularly with games that have a lot of moving parts. In Chapter 1, we mention some Android features that are not supported on the emulator.
- ***Multimedia.*** The apps in the print book use a broad range of Android multimedia capabilities, including graphics, images, frame-by-frame animation and audio. The apps in the online chapters use property animation, video, speech synthesis and speech recognition.
- ***Android Best Practices.*** We adhere to accepted Android best practices, pointing them out in the detailed code walkthroughs. For more information, visit `http://developer.android.com/guide/practices/index.html`.
- ***Java Content in the Appendices Can Be Used With Java SE 6 or Higher.***

- *Java Exception Handling.* We integrate basic exception handling early in the Java content then present a richer treatment in Appendix H; we use exception handling throughout the Android chapters.
- *Classes `Arrays` and `ArrayList`; Collections.* Appendix E covers class `Arrays`—which contains methods for performing common array manipulations—and generic class `ArrayList`—which implements a dynamically resizable array-like data structure. Appendix J introduces Java's generic collections that are used frequently in our Android treatment.
- *Java Multithreading.* Maintaining app responsiveness is a key to building robust Android apps and requires extensive use of Android multithreading. Appendix J introduces multithreading fundamentals so that you can understand our use of the Android `AsyncTask` class in Chapter 8.
- *GUI Presentation.* Appendix I introduces Java GUI development. Android provides its own GUI components, so this appendix presents a few Java GUI components and focuses on nested classes and anonymous inner classes, which are used extensively for event-handling in Android GUIs.

Working with Open-Source Apps

There are numerous free, open-source Android apps available online which are excellent resources for learning Android app development. We encourage you to download open-source apps and read their source code to understand how they work. Throughout the book you'll find programming exercises that ask you to modify or enhance existing open-source apps. Our goal is to give you handles on interesting problems that may also inspire you to create new apps using the same technologies. **Caution: The terms of open source licenses vary considerably.** Some allow you to use the app's source code freely for any purpose, while others stipulate that the code is available for personal use only—not for creating for-sale or publicly available apps. **Be sure to read the licensing agreements carefully. If you wish to create a commercial app based on an open-source app, you should consider having an intellectual property attorney read the license; be aware that these attorneys charge significant fees.**

Pedagogic Features

Syntax Shading. For readability, we syntax shade the code, similar to Eclipse's and Android Studio's use of syntax coloring. Our syntax-shading conventions are as follows:

```
comments appear in gray
constants and literal values appear in bold darker gray
keywords appear in bold black
all other code appears in non-bold black
```

Code Highlighting. We emphasize the key code segments in each program by enclosing them in light gray rectangles.

Using Fonts for Emphasis. We use various font conventions:

- The defining occurrences of key terms appear in **bold** for easy reference.
- On-screen IDE components appear in **bold Helvetica** (e.g., the **File** menu).

- Program source code appears in `Lucida` (e.g., `int x = 5;`).

In this book you'll create GUIs using a combination of visual programming (point and click, drag and drop) and writing code.

We use different fonts when we refer to GUI elements in program code versus GUI elements displayed in the IDE:

- When we refer to a GUI component that we create in a program, we place its class name and object name in a `Lucida` font—e.g., "`Button saveContactButton`."
- When we refer to a GUI component that's part of the IDE, we place the component's text in a **bold Helvetica** font and use a plain text font for the component's type—e.g., "the **File** menu" or "the **Run** button."

Using the > Character. We use the > character to indicate selecting a menu item from a menu. For example, we use the notation **File > New** to indicate that you should select the **New** menu item from the **File** menu.

Source Code. All of the book's source code is available for download from:

```
www.pearsonglobaleditions.com/Deitel
```

Chapter Objectives. Each chapter begins with a list of learning objectives.

Figures. Hundreds of tables, source code listings and screen shots are included.

Software Engineering. We stress program clarity and performance, and concentrate on building well-engineered, object-oriented software.

Self-Review Exercises and Answers. Extensive self-review exercises *and* answers are included for self study.

Exercises with a Current Flair. We've worked hard to create topical Android app-development exercises. You'll develop apps using a broad array of current technologies. All of the Android programming exercises require the implementation of complete apps. You'll be asked to enhance the existing chapter apps, develop similar apps, use your creativity to develop your own apps that use the chapter technologies and build new apps based on open-source apps available on the Internet (and again, **be sure to read and comply with the open-source code-license terms for each app**). The Android exercises also include short-answer fill-in and true/false questions.

In the Java exercises, you'll be asked to recall important terms and concepts; indicate what code segments do; indicate what's wrong with a portion of code; write Java statements, methods and classes; and write complete Java programs.

Index. We include an extensive index for reference. The page number of the defining occurrence of each key term in the book is highlighted in the index in **bold**.

Software Used in *Android How to Program, 2/e*

All the software you'll need for this book is available free for download from the Internet. See the Before You Begin section for the download links.

Documentation. All the Android and Java documentation you'll need to develop Android apps is available free at `http://developer.android.com` and `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. The documentation for Eclipse is available at `www.eclipse.org/documentation`. The documentation for Android Studio is available at `http://developer.android.com/sdk/installing/studio.html`.

Instructor Resources

The following supplements are available to **qualified college instructors only** through Pearson Education's Instructor Resource Center `www.pearsonglobaleditions.com/Deitel`:

- *PowerPoint® slides* containing all the code and figures in the text.
- *Test Item File* of short-answer questions.
- *Solutions Manual* with solutions to the end-of-chapter **short-answer exercises** for *both* the Java and Android content. For the Java content, solutions are provided for *most* of the programming exercises.

  The suggested Android app-development project exercises are *not* typical homework problems. These tend to be *substantial* projects—many of which could require weeks of effort, possibly with students working in teams. ***Selected solutions only*** are provided for these project exercises—these will be available on the Pearson Instructor's Resource Center (IRC) for fall semester 2014 classes. Contact us at `deitel@deitel.com` if you have any questions.

Please do not write to us requesting access to the Pearson Instructor's Resource Center. Access is restricted to qualified college instructors teaching from the book. Instructors may obtain access *only* through their Pearson representatives. If you're not a registered faculty member, contact your Pearson representative.

Before You Begin

For information configuring your computer so that you can develop apps with Java and Android, see the Before You Begin section that follows this Preface.

Acknowledgments

Thanks to Barbara Deitel for long hours devoted to this project—she created all of our Java and Android Resource Centers, and patiently researched hundreds of technical details.

This book was a cooperative effort between the academic and professional divisions of Pearson. We appreciate the guidance, wisdom and energy of Tracy Johnson, Executive Editor, Computer Science. Tracy and her team handle all of our academic textbooks. Carole Snyder recruited the book's academic reviewers and managed the review process. Bob Engelhardt managed the book's publication. We selected the cover art and Marta Samsel designed the cover.

We also appreciate the efforts and 18-year mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of the Pearson Technology Group. Mark and his team handle all of our professional books and LiveLessons video products. Kim Boe-

digheimer recruited and managed the professional reviewers for the Android content. John Fuller manages the production of all of our Deitel Developer Series books.

We'd like to thank Michael Morgano, a former colleague of ours at Deitel & Associates, Inc., now an Android developer at Imerj™, who co-authored the first editions of this book and our book, *iPhone for Programmers: An App-Driven Approach*. Michael is an extraordinarily talented software developer.

Reviewers of the Content from **Android How to Program** *and* **Android for Programmers: An App-Driven Approach** *Recent Editions*

We wish to acknowledge the efforts of our first and second edition reviewers. They scrutinized the text and the code and provided countless suggestions for improving the presentation: Paul Beusterien (Principal, Mobile Developer Solutions), Eric J. Bowden, COO (Safe Driving Systems, LLC), Tony Cantrell (Georgia Northwestern Technical College), Ian G. Clifton (Independent Contractor and Android App Developer, Daniel Galpin (Android Advocate and author of *Intro to Android Application Development*), Jim Hathaway (Application Developer, Kellogg Company), Douglas Jones (Senior Software Engineer, Fullpower Technologies), Charles Lasky (Nagautuck Community College), Enrique Lopez-Manas (Lead Android Architect, Sixt, and Computer Science Teacher at the University of Alcalá in Madrid), Sebastian Nykopp (Chief Architect, Reaktor), Michael Pardo (Android Developer, Mobiata), Ronan "Zero" Schwarz (CIO, OpenIntents), Arijit Sengupta (Wright State University), Donald Smith (Columbia College), Jesus Ubaldo Quevedo-Torrero (University of Wisconsin, Parkside), Dawn Wick (Southwestern Community College) and Frank Xu (Gannon University).

Reviewers of the Content from **Java How to Program** *Recent Editions*

Lance Andersen (Oracle), Soundararajan Angusamy (Sun Microsystems), Joseph Bowbeer (Consultant), William E. Duncan (Louisiana State University), Diana Franklin (University of California, Santa Barbara), Edward F. Gehringer (North Carolina State University), Huiwei Guan (Northshore Community College), Ric Heishman (George Mason University), Dr. Heinz Kabutz (JavaSpecialists.eu), Patty Kraft (San Diego State University), Lawrence Premkumar (Sun Microsystems), Tim Margush (University of Akron), Sue McFarland Metzger (Villanova University), Shyamal Mitra (The University of Texas at Austin), Peter Pilgrim (Consultant), Manjeet Rege, Ph.D. (Rochester Institute of Technology), Manfred Riem (Java Champion, Consultant, Robert Half), Simon Ritter (Oracle), Susan Rodger (Duke University), Amr Sabry (Indiana University), José Antonio González Seco (Parliament of Andalusia), Sang Shin (Sun Microsystems), S. Sivakumar (Astra Infotech Private Limited), Raghavan "Rags" Srinivas (Intuit), Monica Sweat (Georgia Tech), Vinod Varma (Astra Infotech Private Limited) and Alexander Zuev (Sun Microsystems).

As you read the book, we'd sincerely appreciate your comments, criticisms and suggestions for improving the text. Please address all correspondence to:

```
    deitel@deitel.com
```

We'll respond promptly. We really enjoyed writing this book—we hope you enjoy reading it!

Paul Deitel
Harvey Deitel
Abbey Deitel

About the Authors

Paul Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. He holds the Java Certified Programmer and Java Certified Developer certifications, and is an Oracle Java Champion. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses worldwide to clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book/video authors.

Dr. Harvey Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 50 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul Deitel. The Deitels' publications have earned international recognition, with translations published in Simplified Chinese, Traditional Chinese, Korean, Japanese, German, Russian, Spanish, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, academic, government and military clients.

Abbey Deitel, President of Deitel & Associates, Inc., is a graduate of Carnegie Mellon University's Tepper School of Management where she received a B.S. in Industrial Management. Abbey has been managing the business operations of Deitel & Associates, Inc. for 16 years. She has contributed to numerous Deitel & Associates publications and, together with Paul and Harvey, is the co-author of *Android for Programmers: An App-Driven Approach, 2/e*, *iPhone for Programmers: An App-Driven Approach*, *Internet & World Wide Web How to Program, 5/e*, *Visual Basic 2012 How to Program, 6/e* and *Simply Visual Basic 2010, 5/e*.

Deitel® Dive-Into® Series Programming Languages Training

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, mobile app development and Internet and web software technology. The company's training clients include many of the world's largest companies, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including Android app development, Objective-C and iOS app development, Java™, XML®, C++, C, Visual C#®, Visual Basic®, Visual C++®, Python®, object technology, Internet and web programming and a growing list of additional programming and software development courses.

Through its 37-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., publishes leading-edge programming college textbooks and professional books in print and a wide range of electronic formats and *LiveLessons* video courses. Deitel & Associates, Inc. and the authors can be reached at:

```
    deitel@deitel.com
```

To learn more about Deitel's *Dive-Into® Series* Corporate Training curriculum, visit:

```
http://www.deitel.com/training
```

To request a proposal for worldwide on-site, instructor-led training at your organization, e-mail `deitel@deitel.com`.

Individuals wishing to purchase Deitel books and *LiveLessons* video training can do so through `www.deitel.com`. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, please contact your Pearson representative.

Pearson wishes to thank and acknowledge the following people for their work on the Global Edition:

Contributor:
Muthuraj M., Android Developer

Reviewers:
SC Raghavendra SSE, Intuit, India
Manasa S., NMAM Institute of Technology, Nitte, India

Before You Begin

In this section, you'll set up your computer for use with this book. The Android development tools are frequently updated. Before reading this section, check the book's website

```
www.pearsonglobaleditions.com/Deitel
```

to see if we've posted an updated version.

Font and Naming Conventions

We use fonts to distinguish between on-screen components (such as menu names and menu items) and Java code or commands. Our convention is to show on-screen components in a sans-serif bold **Helvetica** font (for example, **Project** menu) and to show file names, Java code and commands in a sans-serif `Lucida` font (for example, the keyword `public` or class `Activity`). When specifying commands to select in menus, we use the **>** notation to indicate a menu item to select. For example, **Window > Preferences** indicates that you should select the **Preferences** menu item from the **Window** menu.

Software and Hardware System Requirements

To develop Android apps you need a Windows®, Linux or Mac OS X system. To view the latest operating-system requirements visit:

```
http://developer.android.com/sdk/index.html
```

and scroll down to the **SYSTEM REQUIREMENTS** heading. We developed the apps in this book using the following software:

- Java SE 7 Software Development Kit
- Android SDK/ADT Bundle based on the Eclipse IDE
- Android SDK versions 4.3 and 4.4

You'll see how to obtain each of these in the next sections.

Installing the Java Development Kit (JDK)

Android requires the *Java Development Kit* (*JDK*) version 7 (JDK 7) or 6 (JDK 6). *We used JDK 7*. To download the JDK for Windows, OS X or Linux, go to

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

You need only the JDK. Choose the 32-bit or 64-bit version based on your computer hardware and operating system. Most recent computers have 64-bit hardware—check your system's specifications. If you have a 32-bit operating system, you must use the 32-bit JDK. Be sure to follow the installation instructions at

```
http://docs.oracle.com/javase/7/docs/webnotes/install/index.html
```

Android Integrated Development Environment (IDE) Options

Google now provides two Android IDE options:

- Android SDK/ADT bundle—a version of the *Eclipse IDE* that comes preconfigured with the latest Android Software Development Kit (SDK) and the latest Android Development Tools (ADT) plugin. At the time of this writing, these were Android SDK version 4.4 and ADT version 22.3.
- Android Studio—Google's new Android IDE based on IntelliJ® IDEA and their preferred future IDE.

The Android SDK/ADT bundle has been widely used in Android app development for several years. Android Studio, introduced in May 2013, is an *early access version* and will be evolving rapidly. For this reason, we'll stay with the widely used Android SDK/ADT bundle in the book, and as online supplements at

```
www.pearsonglobaleditions.com/Deitel
```

we'll provide Android Studio versions of the Chapter 1 Test-Drive section and the Building the GUI section for each app, as appropriate.

Installing the Android SDK/ADT Bundle

To download the Android SDK/ADT bundle, go to

```
http://developer.android.com/sdk/index.html
```

and click the **Download the SDK ADT Bundle** button. When the download completes, extract the ZIP file's contents to your system. The resulting folder has an `eclipse` subfolder containing the Eclipse IDE and an `sdk` subfolder containing the Android SDK. As with the JDK, you can choose a 32-bit or 64-bit version. The Android SDK/ADT bundle 32-bit version should be used with the 32-bit JDK, and the 64-bit version with the 64-bit JDK.

Installing Android Studio

The IDE instructions in the printed book use the Android SDK/ADT bundle. You can also optionally install and use Android Studio. To download Android Studio, go to

```
http://developer.android.com/sdk/installing/studio.html
```

and click the **Download Android Studio** button. When the download completes, run the installer and follow the on-screen instructions to complete the installation. [*Note:* For Android 4.4 development in Android Studio, Android now supports Java SE 7 language features, including the diamond operator, multi-`catch`, `String`s in `switch` and `try`-with-resources.]

Set the Java Compiler Compliance Level and Show Line Numbers

Android does not fully support Java SE 7. To ensure that the book's examples compile correctly, configure Eclipse to produce files that are compatible with Java SE 6 by performing the following steps:

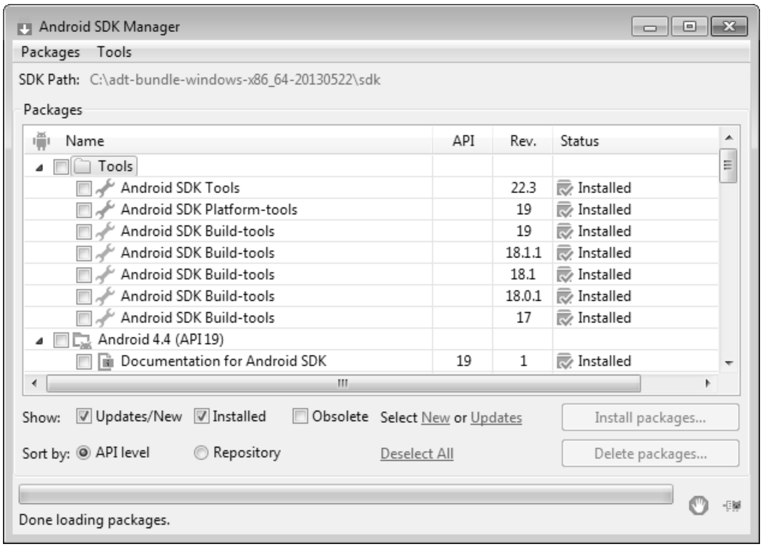
1. Open Eclipse ({} or ●), which is located in the `eclipse` subfolder of the Android SDK/ADT bundle's installation folder.
2. When the **Workspace Launcher** window appears, click **OK**.

3. Select **Window > Preferences** to display the **Preferences** window. On Mac OS X, select **ADT > Preferences....**
4. Expand the **Java** node and select the **Compiler** node. Under **JDK Compliance**, set the **Compiler compliance level** to 1.6 (to indicate that Eclipse should produce compiled code that's compatible with Java SE 6).
5. Expand the **General > Editors** node and select **TextEditors**, then ensure that **Show line numbers** is selected and click **OK**.
6. Close Eclipse.

Android 4.3 SDK

This book's examples were written using the Android 4.3 and 4.4 SDKs. At the time of this writing, 4.4 was the version included with the Android SDK/ADT bundle and Android Studio. You should also install Android 4.3 (and any other versions you might want to support in your apps). To install other Android platform versions, perform the following steps (skipping Steps 1 and 2 if Eclipse is already open):

1. Open Eclipse. Depending on your platform, the icon will appear as or .
2. When the **Workspace Launcher** window appears, click **OK**.
3. On Mac OS X, if you see a window indicating "**Could not find SDK folder '/Users/***YourAccount***/android-sdk-macosx/'**," click **Open Preferences** then **Browse...** and select the `sdk` folder located where you extracted the Android SDK/ADT bundle.
4. Select **Window > Android SDK Manager** to display the **Android SDK Manager** (Fig. 1).



SDK Path: C:\adt-bundle-windows-x86_64-20130522\sdk

Packages

Name	API	Rev.	Status
Tools			
Android SDK Tools		22.3	Installed
Android SDK Platform-tools		19	Installed
Android SDK Build-tools		19	Installed
Android SDK Build-tools		18.1.1	Installed
Android SDK Build-tools		18.1	Installed
Android SDK Build-tools		18.0.1	Installed
Android SDK Build-tools		17	Installed
Android 4.4 (API 19)			
Documentation for Android SDK	19	1	Installed

Show: [✓] Updates/New [✓] Installed [ ] Obsolete Select New or Updates

Sort by: (●) API level ( ) Repository Deselect All

Install packages...

Delete packages...

Done loading packages.

Fig. 1 | **Android SDK Manager** window.

5. The **Android SDK Manager**'s **Name** column shows all of the tools, platform versions and extras (such as APIs for interacting with Google services, like Maps) that you

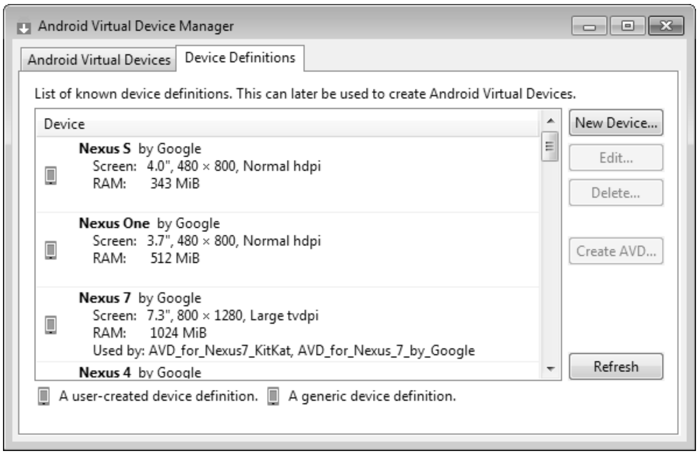
can install. Uncheck the **Installed** checkbox. Then, if any of **Tools**, **Android 4.4 (API19)**, **Android 4.3 (API18)** and **Extras** appear in the **Packages** list, ensure that they're checked and click **Install # packages...** (# is the number of items to be installed) to display the **Choose Packages to Install** window. Most items in the **Extras** node are optional. For this book, you'll need the **Android Support Library** and **Google Play services**. The **Google USB Driver** is necessary for Windows users who wish to test apps on Android devices.]

6. In the **Choose Packages to Install** window, read the license agreements for each item. When you're done, click the **Accept License** radio button, then click the **Install** button. The status of the installation process will be displayed in the **Android SDK Manager** window.

Creating Android Virtual Devices (AVDs)

The **Android emulator**, included in the Android SDK, allows you to test apps on your computer rather than on an actual Android device. This is useful if you're learning Android and don't have access to Android devices, but can be *very* slow, so a real device is preferred if you have one. There are some hardware acceleration features that can improve emulator performance (`developer.android.com/tools/devices/emulator.html#acceleration`). Before running an app in the emulator, you must create an **Android Virtual Device** (**AVD**) which defines the characteristics of the device you want to test on, including the screen size in pixels, the pixel density, the physical size of the screen, size of the SD card for data storage and more. To test your apps for multiple Android devices, you can create AVDs that emulate each unique device. For this book, we use AVDs for Google's Android reference devices—the Nexus 4 phone, the Nexus 7 small tablet and Nexus 10 large tablet—which run unmodified versions of Android. To do so, perform the following steps:

1. Open Eclipse.
2. Select **Window > Android Virtual Device Manager** to display the **Android Virtual Device Manager** window, then select the **Device Definitions** tab (Fig. 2).



List of known device definitions. This can later be used to create Android Virtual Devices.

Device

Nexus S by Google
Screen: 4.0", 480 × 800, Normal hdpi
RAM: 343 MiB

Nexus One by Google
Screen: 3.7", 480 × 800, Normal hdpi
RAM: 512 MiB

Nexus 7 by Google
Screen: 7.3", 800 × 1280, Large tvdpi
RAM: 1024 MiB
Used by: AVD_for_Nexus7_KitKat, AVD_for_Nexus_7_by_Google

Nexus 4 by Google

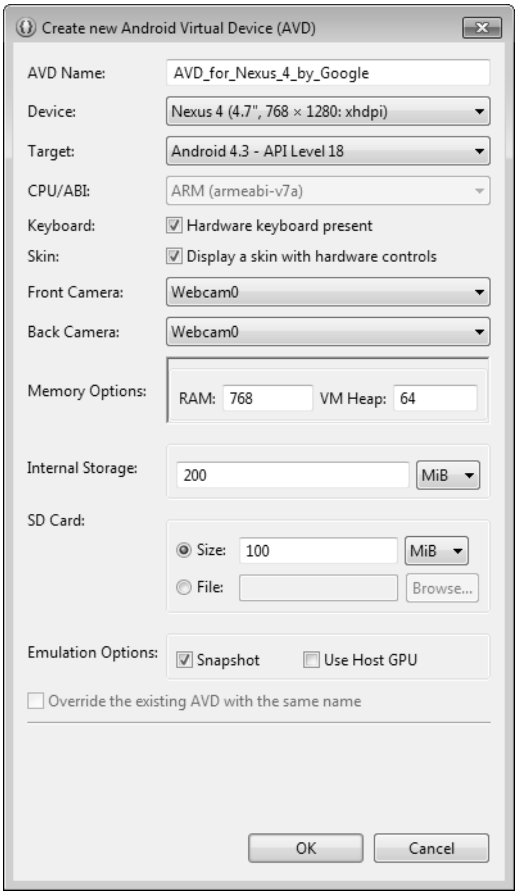
A user-created device definition. A generic device definition.

Fig. 2 | **Android Virtual Device Manager** window.

3. Google provides preconfigured devices that you can use to create AVDs. Select **Nexus 4 by Google**, then click **Create AVD…** to display the **Create new Android Virtual Device (AVD)** window (Fig. 3), then configure the options as shown and click **OK** to create the AVD. If you check **Hardware keyboard present**, you'll be able to use your computer's keyboard to type data into apps that are running in the AVD, but this may prevent the soft keyboard from displaying on the screen. If your computer does not have a camera, you can select **Emulated** for the **Front Camera** and **Back Camera** options. Each AVD you create has many other options specified in its `config.ini`. You can modify this file as described at

   ```
   http://developer.android.com/tools/devices/managing-avds.html
   ```

   to more precisely match the hardware configuration of your device.

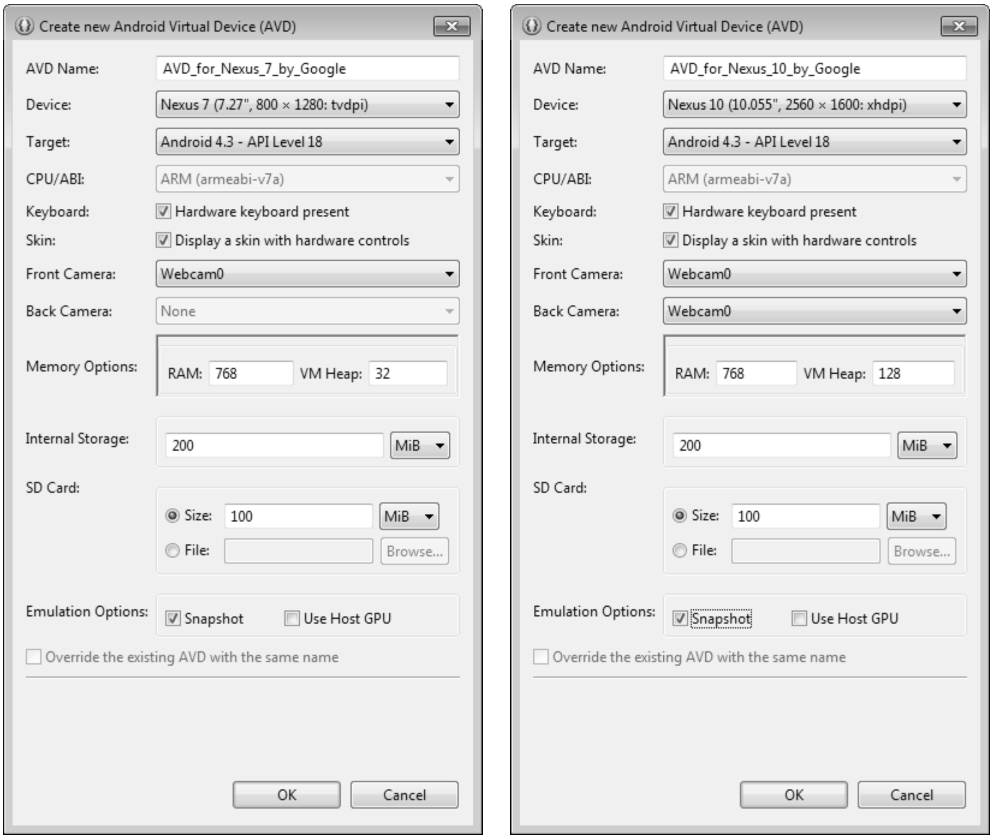

Create new Android Virtual Device (AVD)

AVD Name: AVD_for_Nexus_4_by_Google
Device: Nexus 4 (4.7", 768 × 1280: xhdpi)
Target: Android 4.3 - API Level 18
CPU/ABI: ARM (armeabi-v7a)
Keyboard: [✓] Hardware keyboard present
Skin: [✓] Display a skin with hardware controls
Front Camera: Webcam0
Back Camera: Webcam0
Memory Options: RAM: 768 VM Heap: 64
Internal Storage: 200 MiB
SD Card: (●) Size: 100 MiB ( ) File: Browse...
Emulation Options: [✓] Snapshot [ ] Use Host GPU
[ ] Override the existing AVD with the same name

Fig. 3 | Configuring a Nexus 4 smartphone AVD for Android 4.3.

4. We also configured Android 4.3 AVDs that represent Nexus 7 by Google and Nexus 10 by Google for testing our tablet apps. Their settings are shown in Fig. 4. In

addition, we configured Android 4.4 AVDs for the Nexus 4, Nexus 7 and Nexus 10 with the names: `AVD_for_Nexus_4_KitKat`, `AVD_for_Nexus_7_KitKat`, and `AVD_for_Nexus_10_KitKat`,



Create new Android Virtual Device (AVD)

AVD Name: AVD_for_Nexus_7_by_Google
Device: Nexus 7 (7.27", 800 × 1280: tvdpi)
Target: Android 4.3 - API Level 18
CPU/ABI: ARM (armeabi-v7a)
Keyboard: [✓] Hardware keyboard present
Skin: [✓] Display a skin with hardware controls
Front Camera: Webcam0
Back Camera: None
Memory Options: RAM: 768 VM Heap: 32
Internal Storage: 200 MiB
SD Card: (●) Size: 100 MiB ( ) File: Browse...
Emulation Options: [✓] Snapshot [ ] Use Host GPU
[ ] Override the existing AVD with the same name
OK Cancel

Create new Android Virtual Device (AVD)

AVD Name: AVD_for_Nexus_10_by_Google
Device: Nexus 10 (10.055", 2560 × 1600: xhdpi)
Target: Android 4.3 - API Level 18
CPU/ABI: ARM (armeabi-v7a)
Keyboard: [✓] Hardware keyboard present
Skin: [✓] Display a skin with hardware controls
Front Camera: Webcam0
Back Camera: Webcam0
Memory Options: RAM: 768 VM Heap: 128
Internal Storage: 200 MiB
SD Card: (●) Size: 100 MiB ( ) File: Browse...
Emulation Options: [✓] Snapshot [ ] Use Host GPU
[ ] Override the existing AVD with the same name
OK Cancel

Fig. 4 | Configuring Nexus 7 and Nexus 10 tablet AVDs.

(Optional) Setting Up an Android Device for Development

As we mentioned, testing apps on AVDs can be slow due to AVD performance. If you have an Android device available to you, you should test the apps on that device. In addition, there are some features that you can test only on actual devices. To execute your apps on Android devices, follow the instructions at

```
    http://developer.android.com/tools/device.html
```

If you're developing on Microsoft Windows, you'll also need the Windows USB driver for Android devices. In some cases on Windows, you may also need device-specific USB drivers. For a list of USB driver sites for various device brands, visit:

```
    http://developer.android.com/tools/extras/oem-usb.html
```

Obtaining the Book's Code Examples

The examples for *Android How to Program, 2/e* are available for download at

```
    www.pearsonglobaleditions.com/Deitel
```

If you're not already registered at our website, go to `www.deitel.com` and click the **Register** link. Fill in your information. Registration is free, and we do not share your information with anyone. Please verify that you entered your registration e-mail address correctly—you'll receive a confirmation e-mail with your verification code. *You must click the verification link in the e-mail before you can sign in at* `www.deitel.com` *for the first time*. Configure your e-mail client to allow e-mails from `deitel.com` to ensure that the verification e-mail is not filtered as junk mail. We send only occasional account-management e-mails unless you register separately for our free *Deitel® Buzz Online* e-mail newsletter at

```
    http://www.deitel.com/newsletter/subscribe.html
```

Next, visit `www.deitel.com` and sign in using the **Login** link below our logo in the upper-left corner of the page. Go to `http://www.deitel.com/books/AndroidHTP2/`. Click the **Examples** link to download a ZIP archive file containing the examples to your computer. Double click the ZIP file to unzip the archive, and make note of where you extract the file's contents on your system.

A Note Regarding the Android Development Tools

Google *frequently* updates the Android development tools. This often leads to problems compiling our apps when, in fact, the apps do not contain any errors. If you import one of our apps into Eclipse or Android Studio and it does not compile, there is probably a minor configuration issue. Please contact us by e-mail at `deitel@deitel.com` or by posting a question to:

- Facebook®—`facebook.com/DeitelFan`
- Google+™—`google.com/+DeitelFan`

and we'll help you resolve the issue.

You've now installed all the software and downloaded the code examples you'll need to study Android app development with *Android How to Program, 2/e* and to begin developing your own apps. Enjoy!

1 Introduction to Android

Objectives

In this chapter you'll be introduced to:

- The history of Android and the Android SDK.
- Google Play Store for downloading apps.
- The Android packages used in this book to help you create Android apps.
- Basic object-technology concepts.
- Key software for Android app development, including the Android SDK, the Java SDK, the Eclipse integrated development environment (IDE) and Android Studio.
- Important Android documentation.
- Test-driving an Android drawing app in Eclipse (in the print book) and in Android Studio (online).
- Characteristics of great Android apps.

Outline

1.1 Introduction

Welcome to Android app development! We hope that working with *Android How to Program, 2/e* will be an informative, challenging, entertaining and rewarding experience for you.

This portion of the book is geared toward students with *Java programming experience*. We use only complete working apps, so if you don't know Java but have object-oriented programming experience in another language, such as C#, Objective-C/Cocoa or C++ (with class libraries), you should be able to master the material quickly, learning Java and Java-style object-oriented programming as you learn Android app development. If you do not know Java, we also provide a friendly, rich introduction to it in the book's appendices.

App-Driven Approach

We use an **app-driven approach**—new features are discussed in the context of complete working Android apps, with one app per chapter. For each app, we first describe it, then have you *test-drive* it. Next, we briefly overview the key **Eclipse IDE** (integrated development environment), Java and **Android SDK** (Software Development Kit) technologies we use to implement the app. For apps that require it, we walk through designing the GUI *visually* using Eclipse. Then we provide the complete source-code listing, using line numbers, *syntax shading* and *code highlighting* to emphasize the key portions of the code. We also show one or more screen shots of the running app. Then we do a detailed code walkthrough, emphasizing the new programming concepts introduced in the app. You can download the source code for all of the book's apps from `http://www.deitel.com/books/AndroidHTP2/`.

For each chapter, we also provide **Android Studio** IDE versions of any Eclipse-specific instructions. Because Android Studio is an early access version and will be evolving rapidly, we provide the Android Studio instructions on the book's website

```
    http://www.deitel.com/books/AndroidHTP2
```

This will enable us to keep the instructions up to date.

1.2 Android—The World's Leading Mobile Operating System

Android device sales are growing quickly, creating enormous opportunities for Android app developers.

- The first-generation Android phones were released in October 2008. By October 2013, a Strategy Analytics report showed that Android had 81.3% of the global *smartphone* market share, compared to 13.4% for Apple, 4.1% for Microsoft and 1% for Blackberry.[1]
- According to an IDC report, by the end of the first quarter of 2013 Android had 56.5% of the global *tablet* market share, compared to 39.6% for Apple's iPad and 3.7% for Microsoft Windows tablets.[2]
- As of April 2013, more than 1.5 million Android devices (including smartphones, tablets, etc.) were being activated daily.[3]
- At the time of this writing, there were over *one billion* activated Android devices.[4]
- Android devices now include smartphones, tablets, e-readers, robots, jet engines, NASA satellites, game consoles, refrigerators, televisions, cameras, health-care devices, smartwatches, automobile in-vehicle "infotainment" systems (for controlling the radio, GPS, phone calls, thermostat, etc.) and more.[5]

1.3 Android Features

Openness and Open Source

One benefit of developing Android apps is the openness of the platform. The operating system is *open source* and free. This allows you to view Android's source code and see how its features are implemented. You can also contribute to Android by reporting bugs (see `http://source.android.com/source/report-bugs.html`) or by participating in the Open Source Project discussion groups (`http://source.android.com/community/index.html`). Numerous open-source Android apps from Google and others are available on the Internet

1. `http://blogs.strategyanalytics.com/WSS/post/2013/10/31/Android-Captures-Record-81-Percent-Share-of-Global-Smartphone-Shipments-in-Q3-2013.aspx`.
2. `http://www.idc.com/getdoc.jsp?containerId=prUS24093213`.
3. `http://www.technobuffalo.com/2013/04/16/google-daily-android-activations-1-5-million`.
4. `http://venturebeat.com/2013/09/03/android-hits-1b-activations-and-will-be-called-kitkat-in-next-version`.
5. `http://www.businessweek.com/articles/2013-05-29/behind-the-internet-of-things-is-android-and-its-everywhere`.

(Fig. 1.1). Figure 1.2 shows you where you can get the Android source code, learn about the philosophy behind the open-source operating system and get licensing information.

URL	Description
`http://en.wikipedia.org/wiki/List_of_open_source_Android_applications`	Extensive list of open-source apps, organized by category (e.g., games, communication, emulators, multimedia, security).
`http://developer.android.com/tools/samples/index.html`	Google's sample apps for the Android platform; includes over 60 apps and games such as Lunar Lander, Snake and Tic Tac Toe.
`http://github.com/`	GitHub allows you to share your apps and source code and contribute to others' open-source projects.
`http://sourceforge.net`	SourceForge also allows you to share apps and source code and contribute to others' open-source projects.
`http://f-droid.org/`	Hundreds of free and open-source Android apps including the Adblock Plus advertisement blocker, aMetro public transportation navigation, AnySoftKeyboard (available in several languages), Apollo music player, Chinese Checkers game, DroidWeight weight tracker, Earth Live Wallpaper and more.
`http://blog.interstellr.com/post/39321551640/14-great-android-apps-that-are-also-open-source`	Lists 14 open-source Android apps with links to the code.
`http://www.openintents.org/en/libraries`	Provides nearly 100 open-source libraries that can be used to enhance app capabilities.
`http://www.androidviews.net`	Customized GUI controls for enhancing your app's appearance.
`http://www.stackoverflow.com`	Stack Overflow is a question-and-answer website for programmers. Users can vote on each answer, and the best responses rise to the top.

Fig. 1.1 | Open-source Android app and library resource sites.

Title	URL
Get Android Source Code	`http://source.android.com/source/downloading.html`
Governance Philosophy	`http://source.android.com/about/philosophy.html`
Licenses	`http://source.android.com/source/licenses.html`
FAQs	`http://source.android.com/source/faqs.html`

Fig. 1.2 | Resources and source code for the open-source Android operating system.

The openness of the platform spurs rapid innovation. Unlike Apple's *proprietary* iOS, which is available only on Apple devices, Android is available on devices from dozens of original equipment manufacturers (OEMs) and through numerous telecommunications carriers worldwide. The intense competition among OEMs and carriers benefits customers.

Java

Android apps are developed with Java—one of the world's most widely used programming languages. Java was a logical choice for the Android platform, because it's powerful, free, open source and millions of developers already know it. Experienced Java programmers can quickly dive into Android development, using Google's Android APIs (Application Programming Interfaces) and others available from third parties.

Java is object oriented and has access to extensive class libraries that help you develop powerful apps quickly. GUI programming in Java is *event driven*—in this book, you'll write apps that respond to various user-initiated *events* such as *screen touches*. In addition to directly programming portions of your apps, you'll also use the Eclipse and Android Studio IDEs to conveniently drag and drop predefined objects such as buttons and textboxes into place on your screen, and label and resize them. Using these IDEs, you can create, run, test and debug Android apps quickly and conveniently.

Multi-touch Screen

Android smartphones wrap the functionality of a mobile phone, Internet client, MP3 player, gaming console, digital camera and more into a handheld device with full-color *multi-touch screens*. With the touch of your fingers, you can navigate easily between using your phone, running apps, playing music, web browsing and more. The screen can display a keyboard for typing e-mails and text messages and entering data in apps (some Android devices also have physical keyboards).

Gestures

The multi-touch screens allow you to control the device with *gestures* involving one touch or multiple simultaneous touches (Fig. 1.3).

Gesture name	Physical action	Used to
Touch	Tap the screen once.	Open an app, "press" a button or a menu item.
Double touch	Tap the screen twice.	Zoom in on pictures, Google Maps and web pages. Tap the screen twice again to zoom back out.
Long press	Touch the screen and hold your finger in position.	Select items in a view—for example, checking an item in a list.
Swipe	Touch the screen, then move your finger in the swipe direction and release.	Flip item-by-item through a series, such as photos. A swipe automatically stops at the next item.
Drag	Touch and drag your finger across the screen.	Move objects or icons, or scroll through a web page or list.
Pinch zoom	Pinch two fingers together, or spread them apart.	Zoom in and out on the screen (e.g., resizing text and pictures).

Fig. 1.3 | Some common android gestures.

Built-in Apps

Android devices come with several default apps, which may vary, depending on the device, the manufacturer or the mobile service carrier. These typically include **Phone**, **People**, **Email**, **Browser**, **Camera, Photos, Messaging, Calendar, Play Store, Calculator** and more.

Web Services

Web services are software components stored on one computer that can be accessed by an app (or other software component) on another computer over the Internet. With web services, you can create **mashups**, which enable you to rapidly develop apps by quickly *combining* complementary web services, often from different organizations and possibly other forms of information feeds. For example, 100 Destinations (`www.100destinations.co.uk`) combines the photos and tweets from Twitter with the mapping capabilities of Google Maps to allow you to explore countries around the world through the photos of others.

Programmableweb (`http://www.programmableweb.com/`) provides a directory of over 9,400 APIs and 7,000 mashups, plus how-to guides and sample code for creating your own mashups. Figure 1.4 lists some popular web services. According to Programmableweb, the three most widely used APIs for mashups are Google Maps, Twitter and YouTube.

Web services source	How it's used
Google Maps	Mapping services
Twitter	Microblogging
YouTube	Video search
Facebook	Social networking
Instagram	Photo sharing
Foursquare	Mobile check-in
LinkedIn	Social networking for business
Groupon	Social commerce
Netflix	Movie rentals
eBay	Internet auctions
Wikipedia	Collaborative encyclopedia
PayPal	Payments
Last.fm	Internet radio
Amazon eCommerce	Shopping for books and lots of other products
Salesforce.com	Customer Relationship Management (CRM)
Skype	Internet telephony
Microsoft Bing	Search
Flickr	Photo sharing
Zillow	Real-estate pricing
Yahoo Search	Search
WeatherBug	Weather

Fig. 1.4 | Some popular web services (`http://www.programmableweb.com/apis/directory/1?sort=mashups`).

1.4 Android Operating System

The Android operating system was developed by Android, Inc., which was acquired by Google in 2005. In 2007, the Open Handset Alliance™—which now has 84 company members (`http://www.openhandsetalliance.com/oha_members.html`)—was formed to develop, maintain and evolve Android, driving innovation in mobile technology and improving the user experience while reducing costs.

Android Version Naming Convention

Each new version of Android is named after a dessert, going in alphabetical order (Fig. 1.5).

Android version	Name
Android 1.5	Cupcake
Android 1.6	Donut
Android 2.0–2.1	Eclair
Android 2.2	Froyo
Android 2.3	Gingerbread
Android 3.0–3.2	Honeycomb
Android 4.0	Ice Cream Sandwich
Android 4.1–4.3	Jelly Bean
Android 4.4	KitKat

Fig. 1.5 | Android version numbers and the corresponding names.

1.4.1 Android 2.2 (Froyo)

Android 2.2 (also called **Froyo**, released in May 2010) introduced external storage, allowing you to store apps on an external memory device rather than just in the Android device's internal memory. It also introduced the **Android Cloud to Device Messaging** (**C2DM**) service. **Cloud computing** allows you to use software and data stored in the "cloud"—i.e., accessed on remote computers (or servers) via the Internet and available on demand—rather than having it stored on your desktop, notebook computer or mobile device. Cloud computing gives you the flexibility to increase or decrease computing resources to meet your resource needs at any given time, making it more cost effective than purchasing expensive hardware to ensure that you have enough storage and processing power for occasional peak levels. Android C2DM allows app developers to send data from their servers to their apps installed on Android devices, even when the apps are *not* currently running. The server notifies the apps to contact it directly to receive updated app or user data.[6] C2DM is now deprecated in favor of Google Cloud Messaging.

For information about additional Android 2.2 features—OpenGL ES 2.0 graphics capabilities, the media framework and more—visit `http://developer.android.com/about/versions/android-2.2-highlights.html`.

6. `http://code.google.com/android/c2dm/`.

1.4.2 Android 2.3 (Gingerbread)

Android 2.3 (**Gingerbread**), released later in 2010, added more user refinements, such as a redesigned keyboard, improved navigation capabilities, increased power efficiency and more. It also added several developer features for communications (e.g., technologies that make it easier to make and receive calls from within an app), multimedia (e.g., new audio and graphics APIs) and gaming (e.g., improved performance and new sensors, such as a gyroscope for better motion processing).

One of the most significant new features in Android 2.3 was support for **near-field communication** (**NFC**)—a short-range wireless connectivity standard that enables communication between two devices within a few centimeters. NFC support and features vary by Android device. NFC can be used for payments (for example, touching your NFC-enabled Android device to a payment device on a soda machine), exchanging data such as contacts and pictures, pairing devices and accessories and more.

For a more Android 2.3 developer features, see `http://developer.android.com/about/versions/android-2.3-highlights.html`.

1.4.3 Android 3.0 through 3.2 (Honeycomb)

Android 3.0 (**Honeycomb**) included user-interface improvements specifically for large-screen devices (e.g., tablets), such as a redesigned keyboard for more efficient typing, a visually appealing 3D user interface, easier navigation between screens within an app and more. New Android 3.0 developer features included:

- fragments, which describe portions of an app's user interface and can be combined into one screen or used across multiple screens
- a persistent Action Bar at the top of the screen providing users with options for interacting with apps
- the ability to add large-screen layouts to existing apps designed for small screens to optimize your app for use on different screen sizes
- a visually attractive and more functional user interface, known as "Holo" for its holographic look and feel
- a new animation framework
- improved graphics and multimedia capabilities
- ability to use multicore processor architectures for enhanced performance
- increased Bluetooth support (e.g., enabling an app to determine if there are any connected devices such as headphones or a keyboard)
- and an animation framework for animating user-interface or graphics objects.

For a list of Android 3.0 user and developer features and platform technologies, go to `http://developer.android.com/about/versions/android-3.0-highlights.html`.

1.4.4 Android 4.0 through 4.0.4 (Ice Cream Sandwich)

Android 4.0 (**Ice Cream Sandwich**), released in 2011, merged Android 2.3 (Gingerbread) and Android 3.0 (Honeycomb) into one operating system for use on all Android devices. This allowed you to incorporate into your smartphone apps Honeycomb's features that previously were available only on tablets—the "Holo" user interface, a new launcher (used to customize the device's home screen and launch apps) and more—and easily scale your apps to work on different devices. Ice Cream Sandwich also added several APIs for improved communication between devices, accessibility for users with disabilities (e.g., vision impairments), social networking and more (Fig. 1.6). For a complete list of Android 4.0 APIs, see `http://developer.android.com/about/versions/android-4.0.html`.

Feature	Description
Face detection	Using the camera, compatible devices can determine the positioning of the user's eyes, nose and mouth. The camera can also track the user's eye movement, allowing you to create apps that change perspective, based on where the user is looking.
Virtual camera operator	When filming video of multiple people, the camera will automatically focus on the person who is speaking.
Android Beam	Using NFC, **Android Beam** allows you to touch two Android devices to share content (e.g., contacts, pictures, videos).
Wi-Fi Direct	Wi-Fi P2P (peer-to-peer) APIs allow you to connect multiple Android devices using Wi-Fi. The devices can communicate wirelessly at a greater distance than when using Bluetooth.
Social API	Access and share contact information across social networks and apps (with the user's permission).
Calendar API	Add and share events across multiple apps, manage alerts and attendees and more.
Accessibility APIs	Use the new Accessibility Text-to-Speech APIs to enhance the user experience of your apps for people with disabilities such as vision impairments and more. The explore-by-touch mode allows users with vision impairments to touch anywhere on the screen and hear a voice description of the touched content.
Android@Home framework	Use the **Android@Home framework** to create apps that control appliances in users' homes, such as, thermostats, irrigation systems, networked light bulbs and more.
Bluetooth Health Devices	Create apps that communicate with Bluetooth health devices such as scales, heart-rate monitors and more.

Fig. 1.6 | Some Android Ice Cream Sandwich developer features (`http://developer.android.com/about/versions/android-4.0.html`).

1.4.5 Android 4.1–4.3 (Jelly Bean)

Android Jelly Bean, released in 2012, includes support for external displays, improved security, appearance enhancements (e.g., resizable app widgets and larger app notifications) and performance improvements that make switching between apps and screens more seamless (Fig. 1.7). For the Jelly Bean features list, see `http://developer.android.com/about/versions/jelly-bean.html`.

Feature	Description
Android Beam	You can use Android Beam to easily pair your smartphone or tablet with wireless Bluetooth® speakers or special headphones.
Lock screen widgets	Create widgets that appear on the user's screen when the device is locked, or modify your existing home-screen widgets so that they're also visible when the device is locked.
Photo Sphere	APIs for working with the new panoramic photo features that enable users to take 360-degree photos, similar to those used for Google Maps Street View.
Daydreams	Daydreams are interactive screensavers that are activated when a device is docked or charging. Daydreams can play audio and video and respond to user interactions.
Language support	New features help your apps reach international users, such as bidirectional text (left-to-right or right-to-left), international keyboards, additional keyboard layouts and more.
Developer options	Several new tracking and debugging features help you improve your apps, such as bug reports that include a screen shot and device state information.

Fig. 1.7 | Some Android Jelly Bean features (`http://developer.android.com/about/versions/jelly-bean.html`).

1.4.6 Android 4.4 (KitKat)

Android 4.4 KitKat, released in October 2013, includes several performance improvements that make it possible to run the operating system on all Android devices, including older, memory-constrained devices, which are particularly popular in developing countries.[7]

Enabling more users to update to KitKat will reduce the "fragmentation" of Android versions in the market, which has been a challenge for developers who previously had to design apps to run across multiple versions of the operating system, or limit their potential market by targeting their apps to a specific version of the operating system.

Android KitKat also includes security and accessibility enhancements, improved graphics and multimedia capabilities, memory-use analysis tools and more. Figure 1.8 lists some of the key new KitKat features. For a complete list, see

```
http://developer.android.com/about/versions/kitkat.html
```

Feature	Description
Immersive mode	The status bar at the top of the screen and the menu buttons at the bottom can be hidden, allowing your apps to fill more of the screen. Users can access the status bar by swiping down from the top of the screen, and the system bar (with the back button, home button and recent apps button) by swiping up from the bottom.

Fig. 1.8 | Some Android KitKat features (`http://developer.android.com/about/versions/kitkat.html`). (Part 1 of 2.)

7. `http://techcrunch.com/2013/10/31/android-4-4-kitkat-google/`.

Feature	Description
Printing framework	Build printing functionality into your apps, including locating available printers over Wi-Fi or the cloud, selecting the paper size and specifying which pages to print.
Storage access framework	Create document storage providers that allow users to browse, create and edit files (e.g., documents and images) across multiple apps.
SMS provider	Create SMS (Short Message Service) or MMS (Multimedia Messaging Service) apps using the new SMS provider and APIs. Users can now select their default messaging app.
Transitions framework	The new framework makes it easier to create transition animations.
Screen recording	Record video of your app in action to create tutorials and marketing materials.
Enhanced accessibility	The captioning manager API allows apps to check the user's captioning preferences (e.g., language, text styles and more).
Chromium WebView	Supports the latest standards for displaying web content including HTML5, CSS3 and a faster version of JavaScript.
Step detector and step counter	Create apps that detect whether the user is running, walking or climbing stairs and count the number of steps.
Host Card Emulator (HCE)	HCE enables any app to perform secure NFC transactions (e.g., mobile payments) without the need for a secure element on the SIM card controlled by the wireless carrier.

Fig. 1.8 | Some Android KitKat features (`http://developer.android.com/about/versions/kitkat.html`). (Part 2 of 2.)

1.5 Downloading Apps from Google Play

At the time of this writing, there were over one million apps in **Google Play**, and the number is growing quickly.[8] Figure 1.9 lists some popular free and fee-based apps. You can download apps through the **Play Store** app installed on the device. You can also log into your Google Play account at `http://play.google.com` through your web browser, then specify the Android device on which to install the app. It will then download via the device's WiFi or 3G/4G connection. In Chapter 9, Google Play and App Business Issues, we discuss additional app stores, offering your apps for free or charging a fee, app pricing and more.

Google Play category	Some popular apps in the category
Books and Reference	Kindle, Wikipedia, Audible for Android, Google Play Books
Business	Office Suite Pro 7, Job Search, Square Register, GoToMeeting

Fig. 1.9 | Some popular Android apps in Google Play. (Part 1 of 2.)

8. `en.wikipedia.org/wiki/Google_Play`.

Google Play category	Some popular apps in the category
Comics	ComicRack, Memedroid Pro, Marvel Comics, Comic Strips
Communication	Facebook Messenger, Skype™, GrooVe IP
Education	Duolingo: Learn Languages Free, TED, Mobile Observatory
Entertainment	SketchBook Mobile, Netflix, Fandango® Movies, iFunny :)
Finance	Mint.com Personal Finance, Google Wallet, PayPal
Games: Arcade & Action	Minecraft—Pocket Edition, Fruit Ninja, Angry Birds
Games: Brain & Puzzle	Where's My Water?, Draw Something, Can You Escape
Games: Cards & Casino	Solitaire, Slots Delux, UNO™ & Friends, DH Texas Poker
Games: Casual	Candy Crush Saga, Hardest Game Ever 2, Game Dev Story
Health & Fitness	RunKeeper, Calorie Counter, Workout Trainer, WebMD®
Lifestyle	Zillow Real Estate, Epicurious Recipe App, Family Locator
Live Wallpaper	PicsArt, GO Launcher EX, Beautiful Widgets Pro
Media & Video	MX Player, YouTube, KeepSafe Vault, RealPlayer®
Medical	Epocrates, ICE: In Case of Emergency, Medscape®
Music & Audio	Pandora®, Shazam, Spotify, Ultimate Guitar Tabs & Chords
News & Magazines	Flipboard, Pulse News, CNN, Engadget, Drippler
Personalization	Beautiful Widgets Pro, Zedge™, GO Launcher EX
Photography	Camera ZOOM FX, Photo Grid, InstaPicFrame for Instagram
Productivity	Adobe® Reader®, Dropbox, Google Keep, SwiftKey Keyboard
Shopping	eBay, Amazon Mobile, Groupon, The Coupons App
Social	Facebook®, Instagram, Vine, Twitter, Snapchat, Pinterest
Sports	SportsCenter for Android, NFL '13, Team Stream™
Tools	Titanium Backup PRO, Google Translate, Tiny Flashlight®
Transportation	Uber, Trapster, Lyft, Hailo™, Ulysse Speedometer
Travel & Local	Waze, GasBuddy, KAYAK, TripAdvisor, OpenTable®
Weather	WeatherBug, AccuWeather, The Weather Channel
Widgets	Zillow, DailyHoroscope, Starbucks, Family Locator

Fig. 1.9 | Some popular Android apps in Google Play. (Part 2 of 2.)

1.6 Packages

Android uses a collection of *packages*, which are named groups of related, predefined classes. Some of the packages are Android specific, some are Java specific and some are Google specific. These packages allow you to conveniently access Android OS features and incorporate them into your apps. The Android packages help you create apps that adhere to Android's unique look-and-feel conventions and style guidelines (`http://developer.android.com/design/index.html`). Figure 1.10 lists the packages we discuss in this book. For a complete list of Android packages, see `developer.android.com/reference/packages.html`.

Package	Description
`android.app`	Includes high-level classes in the Android app model. (Chapter 3's **Tip Calculator** app.)
`android.content`	Access and publish data on a device. (Chapter 6's **Cannon Game** app.)
`android.content.res`	Classes for accessing app resources (e.g., media, colors, drawables, etc.), and device-configuration information affecting app behavior. (Chapter 5's **Flag Quiz** app.)
`android.database`	Handling data returned by the content provider. (Chapter 8's **Address Book** app.)
`android.database.sqlite`	SQLite database management for private databases. (Chapter 8's **Address Book** app.)
`android.graphics`	Graphics tools used for drawing to the screen. (Chapter 5's **Flag Quiz** app and Chapter 7's **Doodlz** app.)
`android.hardware`	Device hardware support. (Chapter 7's **Doodlz** app.)
`android.media`	Classes for handling audio and video media interfaces. (Chapter 6's **Cannon Game** app.)
`android.net`	Network access classes. (Chapter 4's **Twitter® Searches** app.)
`android.os`	Operating-systems services. (Chapter 3's **Tip Calculator** app.)
`android.preference`	Working with an app's user preferences. (Chapter 5's **Flag Quiz** app.)
`android.provider`	Access to Android content providers. (Chapter 7's **Doodlz** app.)
`android.support.v4.print`	Android Support Library features for using the Android 4.4 printing framework. (Chapter 7's **Doodlz** app.)
`android.text`	Rendering and tracking text on a device. (Chapter 3's **Tip Calculator** app.)
`android.util`	Utility methods and XML utilities. (Chapter 6's **Cannon Game** app.)
`android.widget`	User-interface classes for widgets. (Chapter 3's **Tip Calculator** app.)
`android.view`	User interface classes for layout and user interactions. (Chapter 4's **Twitter® Searches** app.)
`java.io`	Streaming, serialization and file-system access of input and output facilities. (Chapter 5's **Flag Quiz** app.)
`java.text`	Text formatting classes. (Chapter 4's **Twitter® Searches** app.)
`java.util`	Utility classes. (Chapter 4's **Twitter® Searches** app.)
`android.graphics.drawable`	Classes for display-only elements (e.g., gradients, etc.). (Chapter 5's **Flag Quiz** app.)

Fig. 1.10 | Android and Java packages used in this book, listed with the chapter in which they *first* appear.

1.7 Android Software Development Kit (SDK)

The Android SDK provides the tools you'll need to build Android apps. It's available at no charge through the Android Developers' site. See the Before You Begin section for details on downloading the Android app-development tools you'll need to develop Android apps, including the Java SE, the Android SDK/ADT Bundle (which includes the Eclipse IDE) and the Android Studio IDE.

Android SDK/ADT Bundle
The Android SDK/ADT Bundle—which includes the Eclipse IDE—is the most widely integrated development environment for Android development. Some developers use only a text editor and command-line tools to create Android apps. The Eclipse IDE includes:

- Code editor with support for syntax coloring and line numbering
- Auto-indenting and auto-complete (i.e., type hinting)
- Debugger
- Version control system
- Refactoring support

You'll use Eclipse in Section 1.9 to test-drive the **Doodlz** app. Starting in Chapter 2, **Welcome** App, you'll use Eclipse to build apps.

Android Studio
Android Studio, a new Android IDE based on the JetBrains IntelliJ IDEA Java IDE (`http://www.jetbrains.com/idea/`), was announced in 2013 and is Google's preferred Android IDE of the future. At the time of this writing, Android Studio was available only as an *early access preview*—many of its features were still under development. For each chapter, we also provide Android Studio versions of any Eclipse-specific instructions on the book's website

```
http://www.deitel.com/books/AndroidHTP2
```

To learn more about Android Studio, installing it and migrating from Eclipse, visit `http://developer.android.com/sdk/installing/studio.html`.

Android Development Tools (ADT) Plugin for Eclipse
The **Android Development Tools (ADT) Plugin for Eclipse** (part of the Android SDK/ADT Bundle) allows you to create, run and debug Android apps, export them for distribution (e.g., upload them to Google Play), and more. ADT also includes a visual GUI design tool. GUI components can be dragged and dropped into place to form GUIs without any coding. You'll learn more about ADT in Chapter 2.

The Android Emulator
The **Android emulator**, included in the Android SDK, allows you to run Android apps in a simulated environment within Windows, Mac OS X or Linux, without using an actual Android device. The emulator displays a realistic Android user-interface window. It's particularly useful if you do not have access to Android devices for testing. You should certainly test your apps on a variety of Android devices before uploading them to Google Play.

Before running an app in the emulator, you'll need to create an **Android Virtual Device (AVD)**, which defines the characteristics of the device on which you want to test, including the hardware, system image, screen size, data storage and more. If you want to test your apps for multiple Android devices, you'll need to create separate AVDs to emulate each unique device, or use a service (like `testdroid.com` or `appthwack.com`) that enables you to test on many different devices.

We used the emulator (not an actual Android device) to take most but not all of the Android screen shots for this book. You can reproduce on the emulator most of the

Android gestures (Fig. 1.11) and controls (Fig. 1.12) using your computer's keyboard and mouse. The gestures on the emulator are a bit limited, since your computer probably cannot simulate all the Android hardware features. For example, to test GPS apps in the emulator, you'll need to create files that simulate GPS readings. Also, although you can simulate orientation changes (to *portrait* or *landscape* mode), simulating particular **accelerometer** readings (the accelerometer allows the device to respond to up/down, left/right and forward/backward acceleration) requires features that are not built into the emulator. There is a *Sensor Simulator* available at

```
https://code.google.com/p/openintents/wiki/SensorSimulator
```

that you can use to send simulated sensor information into an AVD to test other sensor features in your apps. Figure 1.13 lists Android functionality that's *not* available on the emulator. You can, however, upload your app to an Android device to test these features. You'll start creating AVDs and using the emulator to develop Android apps in Chapter 2's **Welcome** app.

Gesture	Emulator action
Touch	Click the mouse once. Introduced in Chapter 3's **Tip Calculator** app.
Double touch	Double click the mouse. Introduced in Chapter 6's **Cannon Game** app.
Long press	Click and hold the mouse.
Drag	Click, hold and drag the mouse. Introduced in Chapter 6's **Cannon Game** app.
Swipe	Click and hold the mouse, move the pointer in the swipe direction and release the mouse. Introduced in Chapter 8's **Address Book** app.
Pinch zoom	Press and hold the *Ctrl* (*Control*) key. Two circles that simulate the two touches will appear. Move the circles to the start position, click and hold the mouse and drag the circles to the end position.

Fig. 1.11 | Android gestures on the emulator.

Control	Emulator action
Back	*Esc*
Call/dial button	*F3*
Camera	*Ctrl-KEYPAD_5*, *Ctrl-F3*
End call button	*F4*
Home	*Home* button
Menu (left softkey)	*F2* or *Page Up* button
Power button	*F7*

Fig. 1.12 | Android hardware controls on the emulator (for additional controls, go to `http://developer.android.com/tools/help/emulator.html`). (Part 1 of 2.)

Control	Emulator action
Search	*F5*
* (right softkey)	*Shift-F2* or *Page Down* button
Rotate to previous orientation	*KEYPAD_7*, *Ctrl-F11*
Rotate to next orientation	*KEYPAD_9*, *Ctrl-F12*
Toggle cell networking on/off	*F8*
Volume up button	*KEYPAD_PLUS*, *Ctrl-F5*
Volume down button	*KEYPAD_MINUS*, *Ctrl-F6*

Fig. 1.12 | Android hardware controls on the emulator (for additional controls, go to `http://developer.android.com/tools/help/emulator.html`). (Part 2 of 2.)

Android functionality not available on the emulator
• Making or receiving real phone calls (the emulator allows simulated calls only)
• Bluetooth
• USB connections
• Device-attached headphones
• Determining connected state of the phone
• Determining battery charge or power charging state
• Determining SD card insert/eject
• Sensors (accelerometer, barometer, compass, light sensor, proximity sensor)

Fig. 1.13 | Android functionality not available on the emulator (`http://developer.android.com/tools/devices/emulator.html`).

1.8 Object-Oriented Programming: A Quick Refresher

Android uses object-oriented programming techniques, so in this section we review the basics of object technology. We use all of these concepts in this book.

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. *Objects*, or more precisely the *classes* objects come from, are essentially *reusable* software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating). Software developers are discovering that using a modular, object-oriented design-and-implementation approach can make software development groups much more productive than they could be with earlier popular techniques like "structured programming"—object-oriented programs are often easier to understand, correct and modify.

1.8.1 The Automobile as an Object

To help you understand objects and their contents, let's begin with a simple analogy. Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*. What must happen before you can do this? Well, before you can drive a car, someone has to *design* it. A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal *hides* the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Just as you cannot cook meals in the kitchen of a blueprint, you cannot drive a car's engineering drawings. Before you can drive a car, it must be *built* from the engineering drawings that describe it. A completed car has an *actual* accelerator pedal to make it go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must *press* the pedal to accelerate the car.

1.8.2 Methods and Classes

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **method**. The method houses the program statements that actually perform its tasks. The method hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster. A program unit called a **class** houses the methods that perform the class's tasks. For example, a class that represents a bank account might contain one method to *deposit* money to an account, another to *withdraw* money from an account and a third to *inquire* what the account's current balance is. A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

1.8.3 Instantiation

Just as someone has to *build a car* from its engineering drawings before you can actually drive a car, you must *build an object* of a class before a program can perform the tasks that the class's methods define. The process of doing this is called *instantiation*. An object is then referred to as an **instance** of its class.

1.8.4 Reuse

Just as a car's engineering drawings can be *reused* many times to build many cars, you can *reuse* a class many times to build many objects. **Reuse** of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing*, *debugging* and *performance* tuning. Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

1.8.5 Messages and Method Calls

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is a **method call** that tells a method of the object to perform its task. For example, a program might call a particular bank-account object's *deposit* method to increase the account's balance.

1.8.6 Attributes and Instance Variables

A car, besides having capabilities to accomplish tasks, also has *attributes*, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its *own* attributes. For example, each car knows how much gas is in its own gas tank, but *not* how much is in the tanks of *other* cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank-account object has a *balance attribute* that represents the amount of money in the account. Each bank-account object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank. Attributes are specified by the class's **instance variables**.

1.8.7 Encapsulation

Classes **encapsulate** (i.e., wrap) attributes and methods into objects—an object's attributes and methods are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves. This **information hiding** is crucial to good software engineering.

1.8.8 Inheritance

A new class of objects can be created quickly and conveniently by **inheritance**—the new class absorbs the characteristics of an existing one, possibly customizing them and adding unique characteristics of its own. In our car analogy, a "convertible" certainly *is an* object of the more *general* class "automobile," but more *specifically*, the roof can be raised or lowered.

1.8.9 Object-Oriented Analysis and Design (OOAD)

How will you create the code for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs, but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of 1,000 software developers building the next U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining *what* the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Languages like Java are object oriented. Programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

1.9 Test-Driving the Doodlz App in an Android Virtual Device (AVD)

In this section, you'll run and interact with your first Android app. The **Doodlz** app allows you to drag your fingers on the screen to "paint." You can control the brush sizes and colors using options provided in the app's *options menu*. There is no need to look at the app's code—you'll build the app and study its code in Chapter 7.

The following steps show how to import the app's project into Eclipse and how to test-drive the app in the Nexus 4 Android Virtual Device (AVD) that you set up in the Before You Begin section following the Preface. Later in this section, we'll also discuss how to run the app on a tablet AVD and on an Android device. When the app is running in an AVD, you can create a new painting by "dragging your finger" anywhere on the canvas. You "touch" the screen by using the mouse.

Android SDK/ADT Bundle and Android Studio IDEs

The IDE screen captures in the following steps (and throughout this book) were taken on a computer running Windows 7, the Java SE 7 JDK and the Android SDK/ADT Bundle that you installed in the Before You Begin section. Because Android Studio is an early access version and will be evolving rapidly, we provide the Android Studio instructions for this test-drive on the book's website

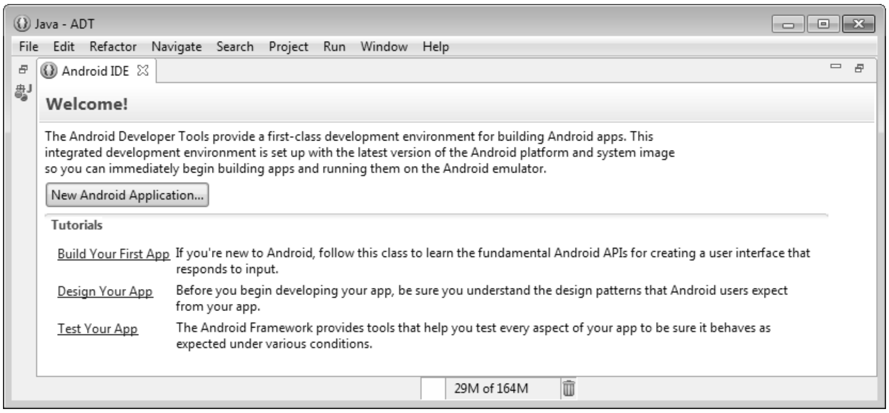
```
www.deitel.com/books/AndroidHTP2
```

This will enable us to update the instructions in response to Google's changes. Both the Android SDK/ADT Bundle and Android Studio use the *same* Android emulator, so once an app is running in an AVD, the steps are identical.

1.9.1 Running the Doodlz App in the Nexus 4 Smartphone AVD

To test-drive the **Doodlz** app, perform the following steps:

1. *Checking your setup.* If you have not done so already, perform the steps specified in the Before You Begin section located after the Preface.
2. *Opening Eclipse.* Open the `eclipse` subfolder of the Android SDK/ADT bundle's installation folder, then double click the Eclipse icon ({} or ●, depending on your platform).
3. *Specifying your workspace location.* When the **Workspace Launcher** window appears, specify where you'd like the apps that you create to be stored, then click **OK**. We used the default location—a folder named `workspace` in your user directory. A **workspace** is a collection of projects, and each project is typically an app or a library that can be shared among apps. Each workspace also has its own settings, such as where various Eclipse subwindows are displayed. You can have many workspaces and switch between them for different development tasks—for example, you could have separate workspaces for Android app development, Java app development and web app development, each with its own custom settings. If this is your first time opening Eclipse, the **Welcome** page (Fig. 1.14) is displayed.


Java - ADT
File Edit Refactor Navigate Search Project Run Window Help
Android IDE
New Android Application...
29M of 164M

Welcome!

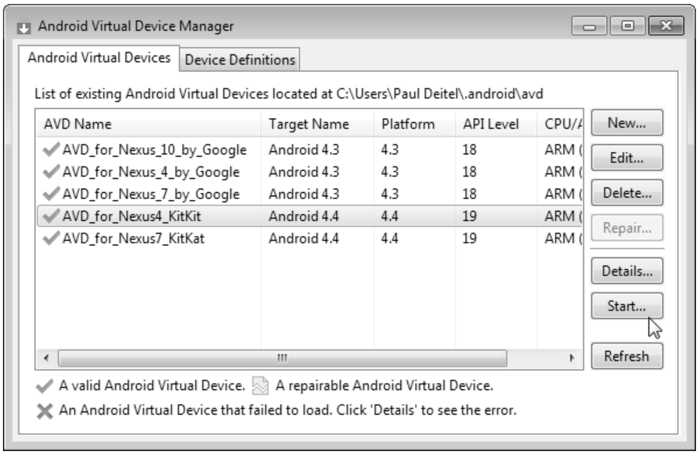
The Android Developer Tools provide a first-class development environment for building Android apps. This integrated development environment is set up with the latest version of the Android platform and system image so you can immediately begin building apps and running them on the Android emulator.

Tutorials

Build Your First App	If you're new to Android, follow this class to learn the fundamental Android APIs for creating a user interface that responds to input.
Design Your App	Before you begin developing your app, be sure you understand the design patterns that Android users expect from your app.
Test Your App	The Android Framework provides tools that help you test every aspect of your app to be sure it behaves as expected under various conditions.

Fig. 1.14 | **Welcome** page in Eclipse.

4. *Launching the Nexus 4 AVD.* For this test-drive, we'll use the Nexus 4 smartphone AVD that you configured for Android 4.4 (KitKat) in the Before You Begin section—in Section 1.9.2, we'll show the app running in a tablet AVD. An AVD can take several minutes to load, so you should launch it in advance of when you intend to use it and keep it running in the background while you're building and testing your apps. To launch the Nexus 4 AVD, select **Window > Android Virtual Device Manager** to display the **Android Virtual Device Manager** dialog (Fig. 1.15). Select the Nexus 4 AVD for Android KitKat and click **Start...**, then click the **Launch** button in the **Launch Options** dialog that appears. You should not attempt to execute the app until the AVD finishes loading. Once the AVD appears as shown in Fig. 1.16, unlock the AVD by dragging the mouse pointer from the lock icon to the edge of the screen.



List of existing Android Virtual Devices located at C:\Users\Paul Deitel\.android\avd

AVD Name	Target Name	Platform	API Level	CPU/A
✓ AVD_for_Nexus_10_by_Google	Android 4.3	4.3	18	ARM (
✓ AVD_for_Nexus_4_by_Google	Android 4.3	4.3	18	ARM (
✓ AVD_for_Nexus_7_by_Google	Android 4.3	4.3	18	ARM (
✓ AVD_for_Nexus4_KitKit	Android 4.4	4.4	19	ARM (
✓ AVD_for_Nexus7_KitKat	Android 4.4	4.4	19	ARM (

✓ A valid Android Virtual Device. A repairable Android Virtual Device.
✗ An Android Virtual Device that failed to load. Click 'Details' to see the error.

Fig. 1.15 | **Android Virtual Device Manager** dialog.

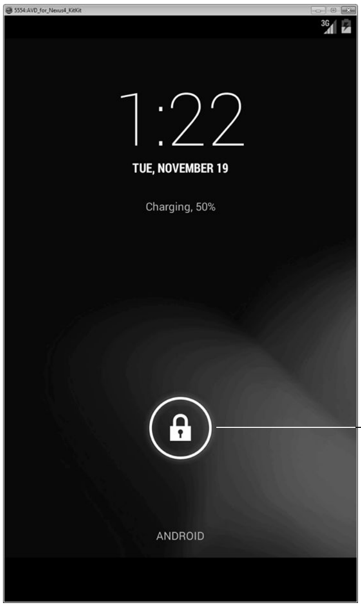

5554:AVD_for_Nexus4_KitKat
1:22
TUE, NOVEMBER 19
Charging, 50%
ANDROID
Drag the mouse pointer from the lock icon to the edge of the screen to unlock the AVD

Fig. 1.16 | Nexus 4 AVD home screen (for Android 4.4) when the AVD finishes loading.

5. ***Importing the* Doodlz *app's project.*** Select **File > Import...** to open the **Import** dialog (Fig. 1.17(a)). Expand the **General** node and select **Existing Projects into Workspace**, then click **Next >** to proceed to the **Import Projects** step (Fig. 1.17(b)). Click the **Browse...** button to the right of the **Select root directory** textbox. In the

a) **Import** dialog

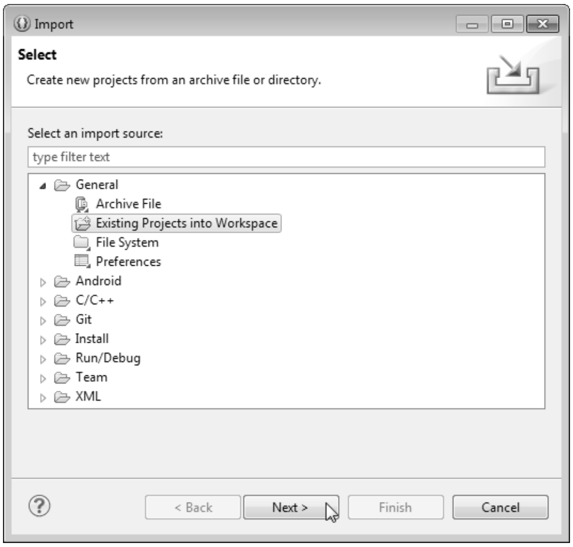


Fig. 1.17 | Importing an existing project. (Part 1 of 2.)

b) **Import** dialog's **Import Projects** step

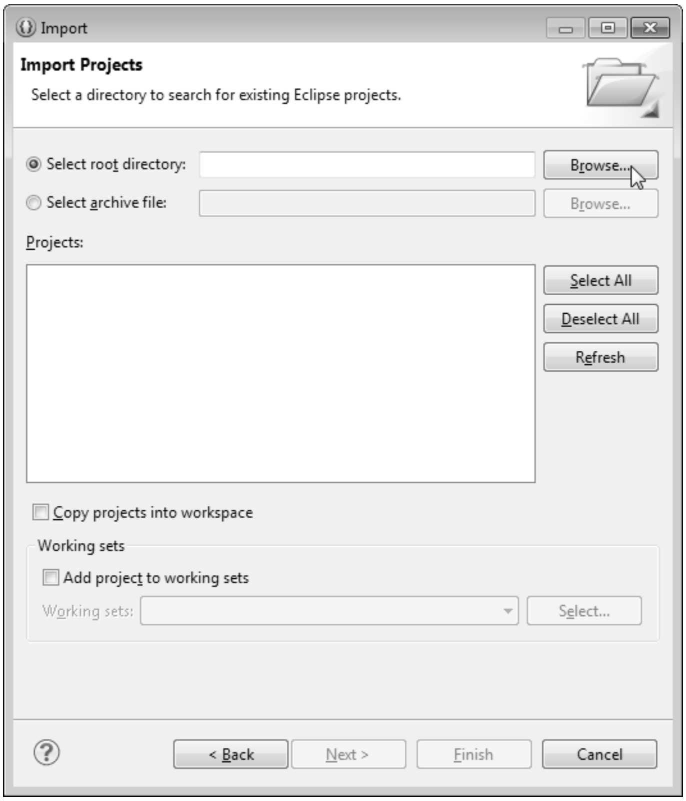


Fig. 1.17 | Importing an existing project. (Part 2 of 2.)

Browse For Folder dialog, locate the **Doodlz** folder in the book's examples folder, select it and click **Open**. Click **Finish** to import the project into Eclipse. The project now appears in the **Package Explorer** window (Fig. 1.18) at the left side of Eclipse. If the **Package Explorer** window is not visible, you can view it by selecting **Window > Show View > Package Explorer**.

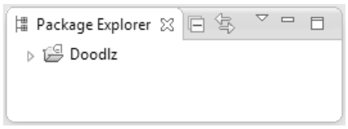


Fig. 1.18 | **Package Explorer** window.

6. *Launching the* **Doodlz** *app.* In Eclipse, right click the **Doodlz** project in the **Package Explorer** window, then select **Run As > Android Application** (Fig. 1.19). This will execute **Doodlz** in the AVD that you launched in Step 4 (Fig. 1.20).

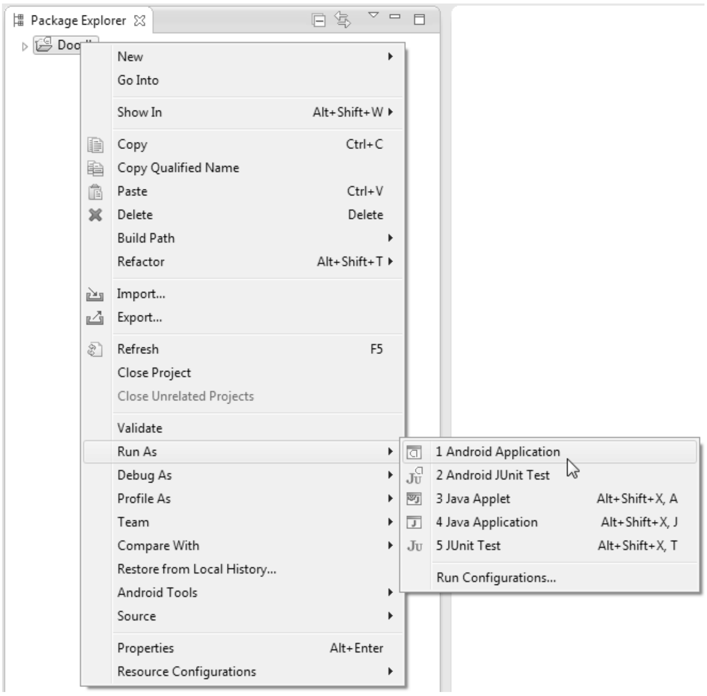


Fig. 1.19 | Launching the **Doodlz** app.

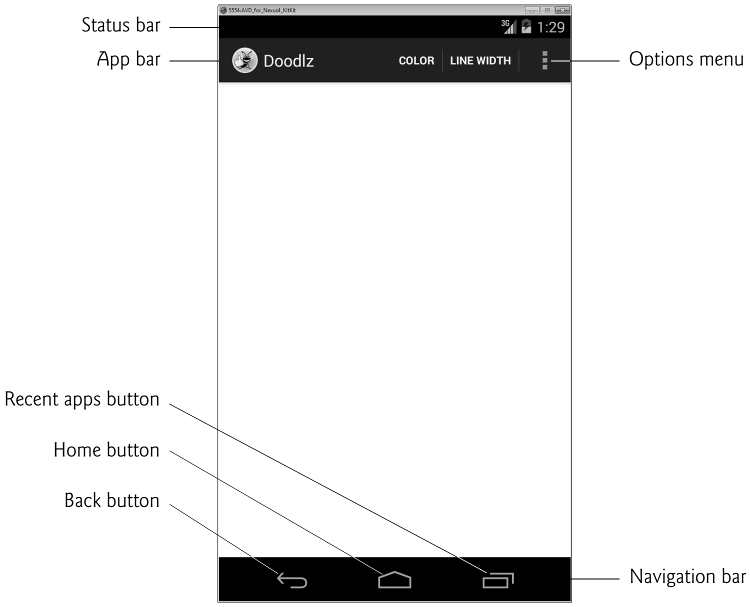


Fig. 1.20 | **Doodlz** app running in the Android Virtual Device (AVD).

7. ***Exploring the AVD and immersive mode.*** At the AVD screen's bottom are various **soft buttons** that appear on the device's touch screen. You touch these (by using the mouse in an AVD) to interact with apps and the Android OS. The *back button* goes back to the app's prior screen, or back to a prior app if you're in the current app's initial screen. The *home button* returns you to the device's home screen. The *recent apps button* allows you to view the recently used apps list, so that you can switch back to them quickly. At the screen's top is the app's *app bar*, which displays the app's icon and name and may contain other app-specific soft buttons—some appear on the app bar (**COLOR** and **LINE WIDTH** in Fig. 1.20) and the rest appear in the app's *options menu* (⋮). The number of options on the app bar depends on the size of the device—we discuss this in Chapter 7. Android 4.4 supports a new *immersive mode* that enables apps to use the entire screen. In this app, you can tap once in the white drawing area to hide the device's status and navigation bars as well as the app's action bar. You can redisplay these by tapping the drawing area again or by swiping from the top edge of the screen.
8. ***Understanding the app's options.*** To display the options that do not appear on the app bar, touch (i.e., click) the options menu (⋮) icon. Figure 1.21(a) shows the action bar and options menu on the Nexus 4 AVD and Fig. 1.21(b) shows them on a Nexus 7 AVD—options shown on the action bar appear in small capital letters. Touching **COLOR** displays a GUI for changing the line color. Touching **LINE WIDTH** displays a GUI for changing the thickness of the line that will be drawn. Touching **Eraser** sets the drawing color to white so that as you draw over colored areas, the color is erased. Touching **Clear** first confirms whether you wish to erase the entire image, then clears the drawing area if you do not cancel the action. Touching **Save Image** saves the image into the device's **Gallery** of images. On Android 4.4, touching **Print** displays a GUI for selecting an available printer so can print your image or save it as a PDF document (the default). You'll explore each of these options momentarily.

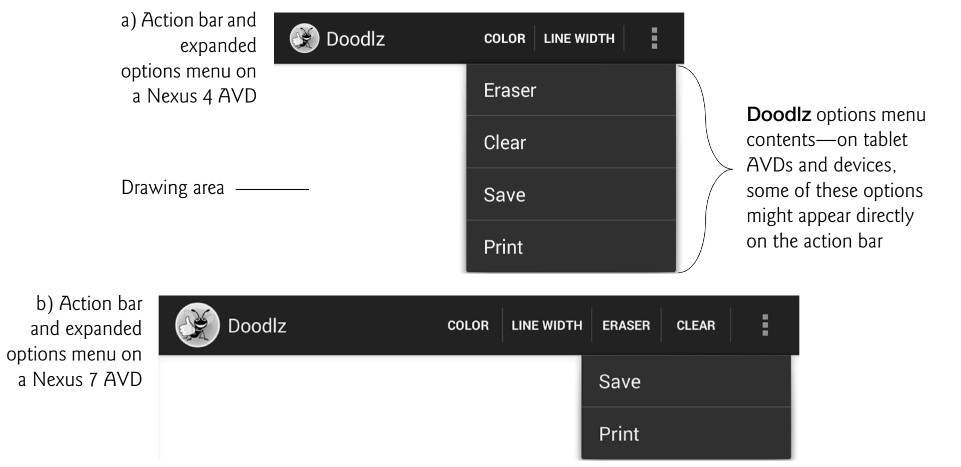
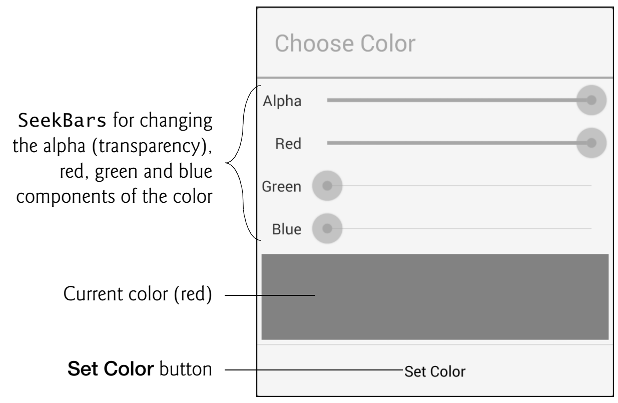

a) Action bar and expanded options menu on a Nexus 4 AVD
Doodlz
COLOR
LINE WIDTH
Eraser
Clear
Save
Print
Drawing area
Doodlz options menu contents—on tablet AVDs and devices, some of these options might appear directly on the action bar
b) Action bar and expanded options menu on a Nexus 7 AVD
Doodlz
COLOR
LINE WIDTH
ERASER
CLEAR
Save
Print

Fig. 1.21 | **Doodlz** options menu expanded.

9. ***Changing the brush color to red.*** To change the brush color, first touch **COLOR** on the action bar to display the **Choose Color** dialog (Fig. 1.22). Colors are defined using the *ARGB color scheme* in which the *alpha* (i.e., *transparency*), red, green and blue components are specified by integers in the range 0–255. For alpha, 0 means *completely transparent* and 255 means *completely opaque*. For red, green and blue, 0 means *none* of that color and 255 means the *maximum amount* of that color. The GUI consists of **Alpha**, **Red**, **Green** and **Blue** `SeekBars` that allow you to select the amount of alpha, red, green and blue in the drawing color. You drag the `SeekBars` to change the color. As you do, the app displays the new color below the `SeekBars`. Select a red color now by dragging the **Red** `SeekBar` to the right as in Fig. 1.22. Touch the **Set Color** button to return to the drawing area.



Choose Color
Alpha
Red
Green
Blue
Set Color
SeekBars for changing the alpha (transparency), red, green and blue components of the color
Current color (red)
Set Color button

Fig. 1.22 | Changing the drawing color to red.

10. ***Changing the line width.*** To change the line width, touch **LINE WIDTH** on the action bar to display the **Choose Line Width** dialog. Drag the `SeekBar` for the line width to the right to thicken the line (Fig. 1.23). Touch the **Set Line Width** button to return to the drawing area.

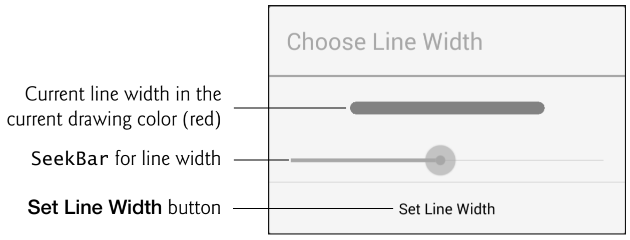


Fig. 1.23 | Changing the line thickness.

11. ***Drawing the flower petals.*** Tap the screen to enter immersive mode, then drag your “finger”—the mouse when using the emulator—on the drawing area to draw flower petals (Fig. 1.24).

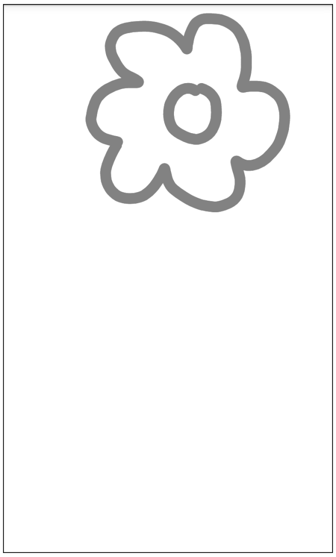
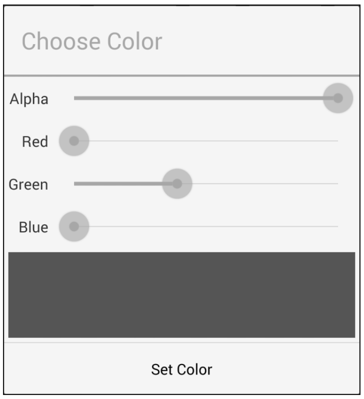


Fig. 1.24 | Drawing flower petals.

12. ***Changing the brush color to dark green.*** Tap the screen to leave immersive mode then touch **COLOR** to display the **Choose Color** dialog. Select a dark green color by dragging the **Green** `SeekBar` to the right and ensuring that the **Red** and **Blue** `SeekBars` are at the far left (Fig. 1.25(a)).

a) Selecting dark green as the drawing color



b) Selecting a thicker line

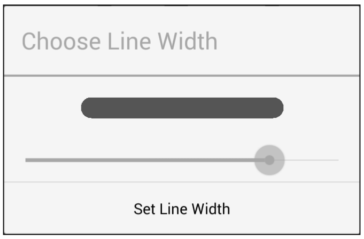


Fig. 1.25 | Changing the color to dark green and making the line thicker.

13. ***Changing the line width and drawing the stem and leaves.*** Touch **LINE WIDTH** to display the **Choose Line Width** dialog. Drag the `SeekBar` for the line width to the right to thicken the line (Fig. 1.25(b)). Tap the screen to re-enter immersive

mode, then draw the flower stem and leaves. Repeat Steps 12 and 13 for a lighter green color and thinner line, then draw the grass (Fig. 1.26).

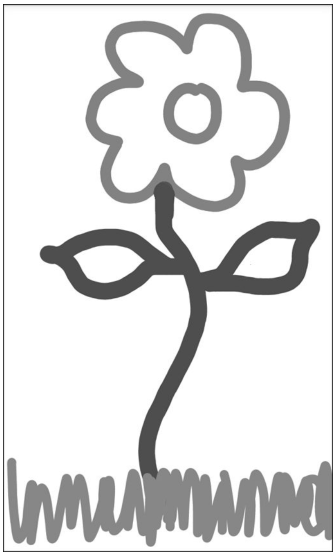


Fig. 1.26 | Drawing the stem and grass.

14. ***Finishing the drawing.*** Tap the screen to exit immersive mode. Next, change the drawing color to blue (Fig. 1.27(a)) and select a narrower line (Fig. 1.27(b)). Then tap the screen to enter immersive mode and draw the raindrops (Fig. 1.28).

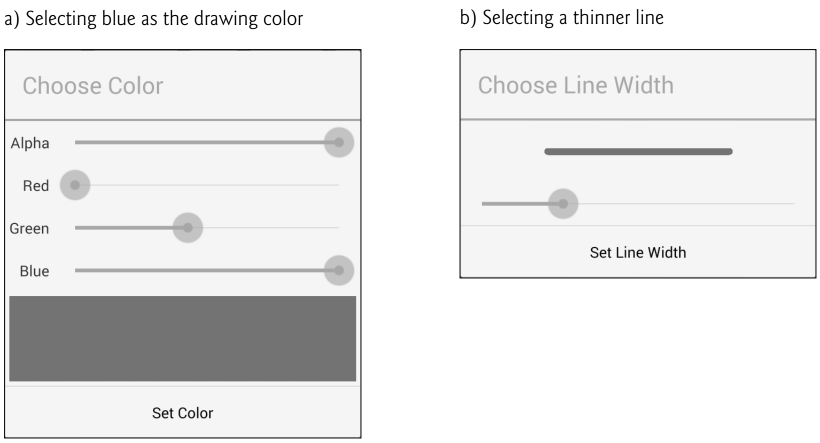


Fig. 1.27 | Changing the line color to blue and narrowing the line.

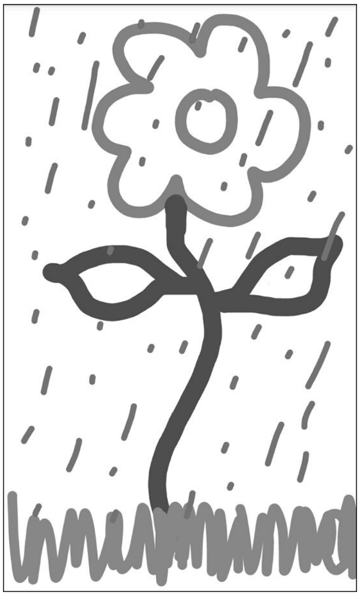


Fig. 1.28 | Drawing the rain in the new line color and line width.

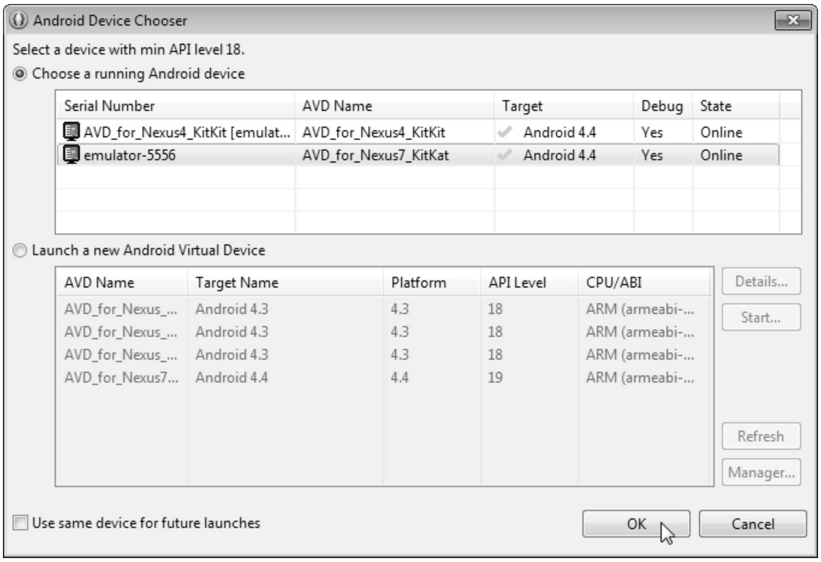
15. ***Saving the image.*** You can save your image to the device's **Gallery** app by selecting **Save** from the options menu ⋮. You can then view this image and others stored on the device by opening the **Gallery** app.
16. ***Printing the image.*** To print the image, select **Print** from the options menu. This displays the print dialog, which allows you to save the image as a PDF document by default. To select a printer, tap **Save as PDF** and select from the available printers. If no printers appear in the list, you may need to configure Google Cloud Print for your printer. For information on this, visit

```
http://www.google.com/cloudprint/learn/
```

17. ***Returning to the home screen.*** You can return to the AVD's home screen by tapping the home (⌂) button on the AVD. To view the drawing in the **Gallery** app touch ⊞ to display the list of apps installed on the AVD. You can then open the **Gallery** app to view the drawing.

1.9.2 Running the Doodlz App in a Tablet AVD

To test the app in a tablet AVD, first launch the AVD by performing Step 4 in Section 1.9.1, but select the Nexus 7 AVD rather than the Nexus 4 AVD. Next, right click the **Doodlz** project in Eclipse's **Package Explorer** window and select **Run As > Android Application**. If multiple AVDs are running when you launch an app, the **Android Device Chooser** dialog (Fig. 1.29) appears so that you can choose the AVD on which to install and execute the app. In this case, both the Nexus 4 and Nexus 7 AVDs were running on our system, so there were two Android virtual devices on which we could possibly run the app. Select the Nexus 7 AVD and click **OK**. This app runs in portrait orientation (the width is less than the height) on phone and small tablet devices. If you run the app on a large tablet AVD (or large tablet device) the app runs in landscape orientation (the width is greater than the height). Figure 1.30 shows the app running in the Nexus 7 AVD. If the AVD is too tall to display on your screen, you can change the AVD's orientation by typing *Ctrl* + *F12* (on a Mac use *fn* + *control* + *F12*). On some keyboards the *Ctrl* key is labeled *Control*.



Android Device Chooser

Select a device with min API level 18.

(●) Choose a running Android device

Serial Number	AVD Name	Target	Debug	State
AVD_for_Nexus4_KitKit [emulat...	AVD_for_Nexus4_KitKit	✓ Android 4.4	Yes	Online
emulator-5556	AVD_for_Nexus7_KitKat	✓ Android 4.4	Yes	Online

( ) Launch a new Android Virtual Device

AVD Name	Target Name	Platform	API Level	CPU/ABI
AVD_for_Nexus_...	Android 4.3	4.3	18	ARM (armeabi-...
AVD_for_Nexus_...	Android 4.3	4.3	18	ARM (armeabi-...
AVD_for_Nexus_...	Android 4.3	4.3	18	ARM (armeabi-...
AVD_for_Nexus7...	Android 4.4	4.4	19	ARM (armeabi-...

☐ Use same device for future launches

Fig. 1.29 | **Android Device Chooser** dialog.

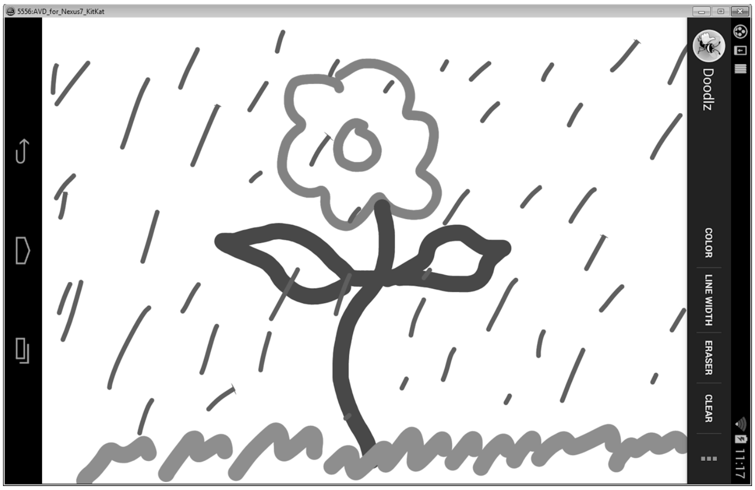


Fig. 1.30 | Drawing in the Nexus 7 AVD.

1.9.3 Running the Doodlz App on an Android Device

If you have an Android device, you can easily execute an app on it for testing purposes.

1. ***Enabling the developer options on the device.*** First, you must enable debugging on the device. To do so, go to the device's **Settings** app, then select **About phone**, (or **About tablet**) locate the **Build number** (at the bottom of the list) and tap it repeatedly until you see the message **You are now a developer** on the screen. This will enable an entry named **Developer options** to the **Settings** app.
2. ***Enabling debugging on the device.*** Return to the **Settings** app, select **Developer options** and ensure that **USB debugging** is checked—this is the default when you first enable the developer options on the device.
3. ***Connecting your device.*** Next, connect the device to your computer via the USB cable that came with your device. If you're a Windows user, recall from the Before You Begin section that you might need to install a USB driver for your device. See the following two web pages for details:

```
developer.android.com/tools/device.html
developer.android.com/tools/extras/oem-usb.html
```

4. ***Running Doodlz on the Android device.*** In Eclipse, right click the **Doodlz** project in the **Package Explorer** window, then select **Run As > Android Application**. If you do not have any AVDs open, but do have an Android device connected, the IDE will automatically install the app on your device and execute it. If you have one or more AVDs open and/or devices connected, the **Android Device Chooser** dialog (Fig. 1.29) is displayed so that you can select the device or AVD on which to install and execute the app.

Preparing to Distribute Apps

When you build apps for distribution via app stores like Google Play, you should test the apps on as many actual devices as you can. Remember that some features can be tested *only* on actual devices. If you don't have many devices available to you, create AVDs that simulate the various devices on which you'd like your app to execute. When you configure each AVD to simulate a particular device, look up the device's specifications online and configure the AVD accordingly. In addition, you can modify the AVD's `config.ini` file as described in the section **Setting hardware emulation options** at

```
developer.android.com/tools/devices/
   managing-avds-cmdline.html#hardwareopts
```

This file contains options that are not configurable via the **Android Virtual Device Manager**. Modifying these options allows you to more precisely match the hardware configuration of an actual device.

1.10 Building Great Android Apps

With over 800,000 apps in Google Play,[9] how do you create an Android app that people will find, download, use and recommend to others? Consider what makes an app fun, useful, in-

9. `http://www.pureoxygenmobile.com/how-many-apps-in-each-app-store/`.

teresting, appealing and enduring. A clever app name, an attractive icon and an engaging description might lure people to your app on Google Play or one of the many other Android app marketplaces. But once users download the app, what will make them use it regularly and recommend it to others? Figure 1.31 shows some characteristics of great apps.

Characteristics of great apps

Great Games

- Entertaining and fun.
- Challenging.
- Progressive levels of difficulty.
- Show your scores and use leaderboards to record high scores.
- Provide audio and visual feedback.
- Offer single-player, multiplayer and networked versions.
- Have high-quality animations.
- Offloading input/output and compute-intensive code to separate threads of execution to improve interface responsiveness and app performance.
- Innovate with augmented reality technology—enhancing a real-world environment with virtual components; this is particularly popular with video-based apps.

Useful Utilities

- Provide useful functionality and accurate information.
- Increase personal and business productivity.
- Make tasks more convenient (e.g., maintaining a to-do list, managing expenses).
- Make the user better informed.
- Provide topical information (e.g., the latest stock prices, news, severe storm warnings, traffic updates).
- Use location-based services to provide local services (e.g., coupons for local businesses, best gas prices, food delivery).

General Characteristics

- Up-to-date with the latest Android features, but compatible with multiple Android versions to support the widest possible audience.
- Work properly.
- Bugs are fixed promptly.
- Follow standard Android app GUI conventions.
- Launch quickly.
- Are responsive.
- Don't require too much memory, bandwidth or battery power.
- Are novel and creative.
- Enduring—something that your users will use regularly.
- Use professional-quality icons that will appear in Google Play and on the user's device.

Fig. 1.31 | Characteristics of great apps. (Part 1 of 2.)

Characteristics of great apps
General Characteristics (cont.)
• Use quality graphics, images, animations, audio and video.
• Are intuitive and easy to use (don't require extensive help documentation).
• Accessible to people with disabilities (`http://developer.android.com/guide/topics/ui/accessibility/index.html`).
• Give users reasons and a means to tell others about your app (e.g., you can give users the option to post their game scores to Facebook or Twitter).
• Provide additional content for content-driven apps (e.g., game levels, articles, puzzles).
• Localized (Chapter 2) for each country in which the app is offered (e.g., translate the app's text and audio files, use different graphics based on the locale, etc.).
• Offer better performance, capabilities and ease-of-use than competitive apps.
• Take advantage of the device's built-in capabilities.
• Do not request excessive permissions.
• Are designed to run optimally across a broad variety of Android devices.
• Future-proofed for new hardware devices—specify the exact hardware features your app uses so Google Play can filter and display it for only compatible devices (`http://android-developers.blogspot.com/2010/06/future-proofing-your-app.html`).

Fig. 1.31 | Characteristics of great apps. (Part 2 of 2.)

1.11 Android Development Resources

Figure 1.32 lists some of the key documentation from the Android Developer site. As you dive into Android app development, you may have questions about the tools, design issues, security and more. There are several Android developer newsgroups and forums where you can get the latest announcements or ask questions (Fig. 1.33). Figure 1.34 lists several websites where you'll find Android development tips, videos and resources.

Title	URL
App Components	`http://developer.android.com/guide/components/index.html`
Using the Android Emulator	`http://developer.android.com/tools/devices/emulator.html`
Package Index	`http://developer.android.com/reference/packages.html`
Class Index	`http://developer.android.com/reference/classes.html`
Android Design	`http://developer.android.com/design/index.html`

Fig. 1.32 | Key online documentation for Android developers. (Part 1 of 2.)

Title	URL
Data Backup	`http://developer.android.com/guide/topics/data/backup.html`
Security Tips	`http://developer.android.com/training/articles/security-tips.html`
Managing Projects from Eclipse with ADT	`http://developer.android.com/guide/developing/projects/projects-eclipse.html`
Getting Started with Android Studio	`http://developer.android.com/sdk/installing/studio.html`
Debugging	`http://developer.android.com/tools/debugging/index.html`
Tools Help	`http://developer.android.com/tools/help/index.html`
Performance Tips	`http://developer.android.com/training/articles/perf-tips.html`
Keeping Your App Responsive	`http://developer.android.com/training/articles/perf-anr.html`
Launch Checklist (for Google Play)	`http://developer.android.com/distribute/googleplay/publish/preparing.html`
Get Started with Publishing	`http://developer.android.com/distribute/googleplay/publish/register.html`
Managing Your App's Memory	`http://developer.android.com/training/articles/memory.html`
Google Play Developer Distribution Agreement	`http://play.google.com/about/developer-distribution-agreement.html`

Fig. 1.32 | Key online documentation for Android developers. (Part 2 of 2.)

Title	Subscribe	Description
Android Discuss	*Subscribe using Google Groups:* `android-discuss` *Subscribe via e-mail:* `android-discuss-subscribe@googlegroups.com`	A general Android discussion group where you can get answers to your app-development questions.
Stack Overflow	`http://stackoverflow.com/questions/tagged/android`	Use this list for beginner-level Android app-development questions, including getting started with Java and Eclipse, and questions about best practices.
Android Developers	`http://groups.google.com/forum/?fromgroups#!forum/android-developers`	Experienced Android developers use this list for troubleshooting apps, GUI design issues, performance issues and more.

Fig. 1.33 | Android newsgroups and forums. (Part 1 of 2.)

Title	Subscribe	Description
Android Forums	`http://www.androidforums.com`	Ask questions, share tips with other developers and find forums targeting specific Android devices.

Fig. 1.33 | Android newsgroups and forums. (Part 2 of 2.)

Android development tips, videos and resources	URL
Sample Android apps from Google	`http://code.google.com/p/apps-for-android/`
O'Reilly article, "Ten Tips for Android Application Development"	`http://answers.oreilly.com/topic/862-ten-tips-for-android-application-development/`
Bright Hub™ website for Android programming tips and how-to guides	`http://www.brighthub.com/mobile/google-android.aspx`
The Android Developers Blog	`http://android-developers.blogspot.com/`
The Sprint® Application Developers Program	`http://developer.sprint.com/site/global/develop/mobile_platforms/android/android.jsp`
HTC's Developer Center for Android	`http://www.htcdev.com/`
The Motorola Android development site	`http://developer.motorola.com/`
Top Android Users on Stack Overflow	`http://stackoverflow.com/tags/android/topusers`
AndroidDev Weekly Newsletter	`http://androiddevweekly.com/`
Chet Haase's Codependent blog	`http://graphics-geek.blogspot.com/`
Cyril Mottier's Android blog	`http://cyrilmottier.com/`
Romain Guy's Android blog	`http://www.curious-creature.org/category/android/`
Android Developers Channel on YouTube®	`http://www.youtube.com/user/androiddevelopers`
Android Video Playlists	`http://developer.android.com/develop/index.html`
What's New in Android Developer Tools	`http://www.youtube.com/watch?v=lmv1dTnhLH4`
Google I/O 2013 Developer Conference session videos	`http://developers.google.com/events/io/sessions`

Fig. 1.34 | Android development tips, videos and resources.

1.12 Wrap-Up

This chapter presented a brief history of Android and discussed its functionality. We provided links to some of the key online documentation and to the newsgroups and forums you can use to connect with the developer community. We discussed features of the Android operating system and provided links to some popular free and fee-based apps on Google Play. We introduced the Java, Android and Google packages that enable you to use the hardware and software functionality you'll need to build a variety of Android apps. You'll use many of these packages in this book. We also discussed Java programming and the Android SDK. You learned the Android gestures and how to perform each on an Android device and on the emulator. We provided a quick refresher on basic object-technology concepts, including classes, objects, attributes and behaviors. You test-drove the **Doodlz** app on the Android emulator for both smartphone and tablet AVDs. In the next chapter, you'll build your first Android app using only visual programming techniques. The app will display text and two images. You'll also learn about Android accessibility and internationalization.

Self-Review Exercises

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

a) App developers can send data from their servers to their apps installed on Android devices even if the apps are *not* running currently using __________.

b) __________ is a short-range wireless connectivity standard that allows communication between two devices within a few centimeters.

c) __________ describe portions of an app's user interface, which can be combined into one screen or used across multiple screens.

d) With web services, you can create __________, which enable you to rapidly develop apps by quickly combining complementary web services, often from different organizations and possibly other forms of information feeds.

e) Android uses a collection of __________, which are named groups of related, predefined classes.

f) The __________, included in the Android SDK, allows you to run Android apps in a simulated environment within Windows, Mac OS X or Linux.

g) Almost any noun can be reasonably represented as a software object in terms of __________ (e.g., name, color and size) and behaviors (e.g., calculating, moving and communicating).

h) Using NFC __________ allows you to touch two Android devices to share content.

i) You send messages to an object. Each message is a(n) __________ that tells a method of the object to perform its task.

1.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Android 2.2 introduced external storage, which allows one to store apps on an external memory device.

b) Cloud computing allows one to use software and data stored in the local machines.

c) Java is neither object-oriented nor has access to extensive class libraries that help you develop powerful apps quickly.

d) Attributes are specified by the class's methods.

e) Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves.

1.3 Fill in the blanks in each of the following statements (based on Section 1.8):

a) Objects have the property of _______—although objects communicate with one another, they're normally not allowed to know how other objects are implemented.
b) The _______ that objects come from are essentially reusable software components; they include attributes and behaviors.
c) The process of analyzing and designing a system from an object-oriented point of view is called _______.
d) With _______, new classes of objects are derived by absorbing characteristics of existing classes, then adding unique characteristics of their own.
e) The size, shape, color and weight of an object are considered _______ of the object's class.
f) A class that represents a bank account might contain one _______ to deposit money to an account, another to withdraw money from an account and a third to inquire what the account's current balance is.
g) You must build an object of a class before a program can perform the tasks that the class's methods define—this process is called _______.
h) The balance of a bank account class is an example of a(n) _______ of that class.
i) Your project's requirements define what the system is supposed to do and your design specifies _______ the system should do it.

Answers to Self-Review Exercises

1.1 a) Android C2DM. b) Near-field communication (NFC). c) Fragments. d) mashups. e) packages. f) Android emulator. g) attributes. h) Android Beam. i) method call.

1.2 a) True. b) False. It allows you to use software and data stored in the "cloud". c) False. Java is object-oriented and has access to extensive class libraries. d) False. Attributes are specified by the class's instance variables. e) True.

1.3 a) information hiding. b) classes. c) object-oriented analysis and design (OOAD). d) inheritance. e) attributes. f) method. g) instantiation. h) attribute. i) how.

Exercises

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

a) Android apps are developed with __________—one of the world's most widely used programming language, a logical choice because it's powerful, free and open source.
b) __________ are software components stored on one computer that can be accessed by an app (or other software component) on another computer over the Internet.
c) Android version 2.3 is also known as __________.
d) Touching the screen, moving your finger in a direction and releasing it generates a __________ gesture.
e) Before running an app in the emulator, you'll need to create an __________, which defines the characteristics of the device on which you want to test, including the hardware, system image, screen size, data storage and more.
f) Performing a task in a program requires a __________ which houses the program statements that actually perform its tasks.
g) You must build an object of a class before a program can perform the tasks that the class's methods define. The process of doing this is called __________.
h) __________ helps you build more reliable and effective systems, because existing classes and components often have gone through extensive testing, debugging and performance tuning.

i) Classes __________ (i.e., wrap) attributes and methods into objects—an object's attributes and methods are intimately related.
j) A new class of objects can be created quickly and conveniently by __________—the new class absorbs the characteristics of an existing one, possibly customizing them and adding unique characteristics of its own.
k) Unlike actual buttons on a device, __________ buttons appear on the device's touch screen.
l) Colors are defined using the RGBA color scheme in which the red, green, blue and __________ components are specified by integers in the range 0–255.

1.5 State whether each of the following is *true* or *false*. If *false*, explain why.
a) The vast majority of Android development is done in C++.
b) Microsoft Visual Studio is the recommended integrated development environment for Android development, though developers may also use a text editor and command-line tools to create Android apps.
c) Reuse helps you build more reliable systems as existing classes and components have often gone through extensive testing, debugging and performance tuning.
d) An object has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class.

1.6 One of the most common objects is a car. Discuss how each of the following terms and concepts applies to the notion of a car: object, attributes, behaviors, class, inheritance (consider, for example, an automatic car), messages, encapsulation, and information hiding.

2 Welcome App

Objectives

In this chapter you'll:

- Learn the basics of the Android Developer Tools (the Eclipse IDE and the ADT Plugin), which you'll use to write, test and debug your Android apps.
- Use the IDE to create a new app project.
- Design a graphical user interface (GUI) visually (without programming) using the IDE's **Graphical Layout** editor.
- Display text and two images in a GUI.
- Edit the properties of GUI components.
- Build and launch an app in the Android emulator.
- Make the app more accessible to visually impaired people by specifying strings for use with Android's TalkBack and Explore-by-Touch features.
- Support internationalization so your app can display strings localized in different languages.

Outline

2.1 Introduction

In this chapter, *without writing any code* you'll build the **Welcome** app that displays a welcome message and two images. You'll use the *Android Developer Tools IDE* to create an app that runs on Android phones. In later chapters you'll also create apps that run on tablets or on both phones and tablets. You'll create a simple Android app (Fig. 2.1) using the IDE's

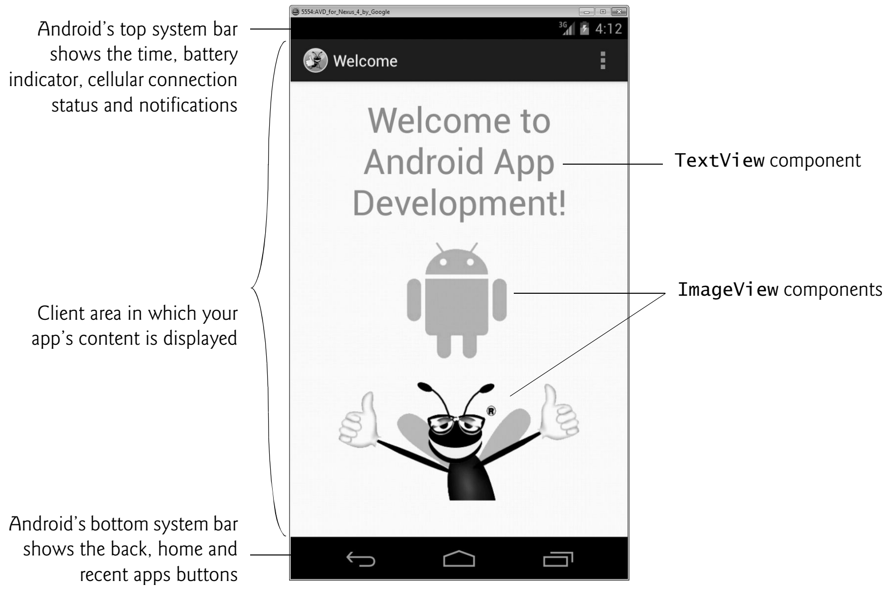

Android's top system bar shows the time, battery indicator, cellular connection status and notifications
Client area in which your app's content is displayed
Android's bottom system bar shows the back, home and recent apps buttons
5554:AVD_for_Nexus_4_by_Google
4:12
Welcome
Welcome to Android App Development!
TextView component
ImageView components

Fig. 2.1 | **Welcome** app running in the Android emulator.

Graphical Layout editor, which allows you to build GUIs using *drag-and-drop* techniques. You'll execute your app in the Android *emulator* (and on an Android phone, if you have one available). Finally, you'll learn how to make the app more *accessible* for people with disabilities and how to *internationalize* it to display strings *localized* in different languages. On the book's website—`http://www.deitel.com/books/AndroidHTP2`—we provide an *Android Studio IDE* version of this chapter. This chapter assumes that you've read the Preface, Before You Begin and Section 1.9.

2.2 Technologies Overview

This section introduces the technologies you'll learn in this chapter.

2.2.1 Android Developer Tools IDE

This chapter introduces the *Android Developer Tools IDE*. You'll use it to create a new project (Section 2.3). As you'll see, the IDE creates a default GUI that contains the text `"Hello world!"` You'll then use the IDE's **Graphical Layout** editor and **Properties** window to visually build a simple graphical user interface (GUI) consisting of text and two images (Section 2.5).

2.2.2 `TextView`s and `ImageView`s

This app's text is displayed in a **`TextView`** and its pictures are displayed in **`ImageView`s**. The default GUI created for this app contains a `TextView`, which you'll modify by using the IDE's **Properties** window to configure various options, such as the `TextView`'s text, font size and font color (Section 2.5.3). Next, you'll use the **Graphical Layout** editor's **Palette** of GUI controls to drag and drop `ImageView`s onto the GUI (Section 2.5.4).

2.2.3 App Resources

It's considered good practice to define all strings and numeric values in resource files that are placed in the subfolders of a project's `res` folder. You'll learn in Section 2.5.3 how to create resources for strings (such as the text on a `TextView`) and measurements (such as a font's size). You'll also learn how to use a built-in Android color resource to specify the `TextView`'s font color.

2.2.4 Accessibility

Android contains many *accessibility* features to help people with various disabilities use their devices. For example, people with visual and physical disabilities can use Android's **TalkBack** to allow a device to speak screen text or text that you provide to help them understand the purpose and contents of a GUI component. Android's **Explore by Touch** enables the user to touch the screen to hear TalkBack speak what's on the screen near the touch. Section 2.7 shows how to enable these features and how to configure your app's GUI components for accessibility.

2.2.5 Internationalization

Android devices are used worldwide. To reach the largest possible audience with your apps, you should consider customizing them for various *locales* and spoken languages—this is known as **internationalization.** Section 2.8 shows how to provide Spanish text for the **Welcome** app's `TextView` and the `ImageViews`' accessibility strings, then shows how to test the app on an AVD configured for Spanish.

2.3 Creating an App

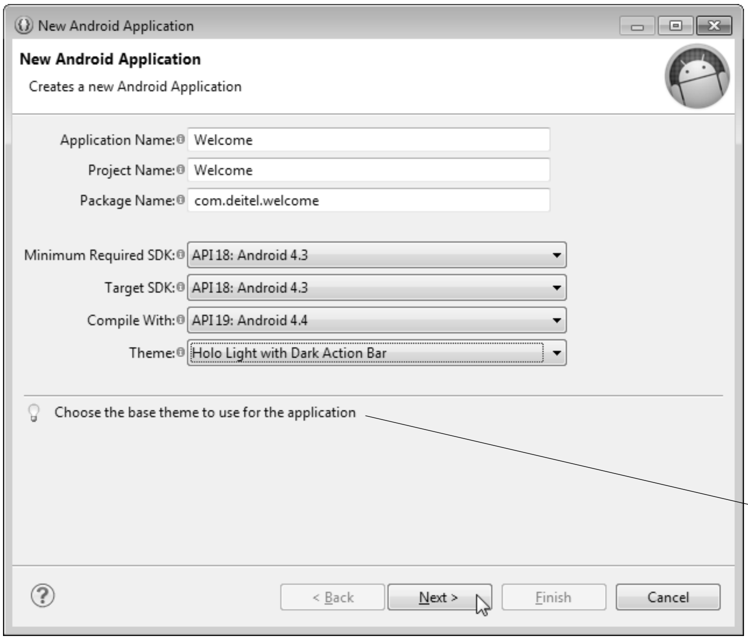
This book's examples were developed using the versions of the Android Developer Tools (version 22.x) and the Android SDK (versions 4.3 and 4.4) that were current at the time of this writing. We assume that you've read the Before You Begin section, and set up the Java SE Development Kit (JDK) and the Android Developer Tools IDE that you used in the test-drive in Section 1.9. This section shows you how to use the IDE to create a new project. We'll introduce additional features of the IDE throughout the book.

2.3.1 Launching the Android Developer Tools IDE

To launch the IDE, open the *Android SDK/ADT bundle* installation folder's `eclipse` subfolder, then double click the Eclipse icon (or), depending on your platform). When you start the IDE for the first time, the **Welcome** page (shown originally in Fig. 1.14) is displayed. If it is not displayed, select **Help > Android IDE** to display it.

2.3.2 Creating a New Project

A **project** is a group of related files, such as code files and images that make up an app. To create an app, you must first create its project. To do so, click the **New Android Application...** button on the **Welcome** page to display the **New Android Application dialog** (Fig. 2.2). You can also do this by selecting **File > New > Android Application Project** or by clicking the **New** () toolbar button's drop-down list and selecting **Android Application Project**.


New Android Application

New Android Application
Creates a new Android Application

Application Name: Welcome
Project Name: Welcome
Package Name: com.deitel.welcome

Minimum Required SDK: API 18: Android 4.3
Target SDK: API 18: Android 4.3
Compile With: API 19: Android 4.4
Theme: Holo Light with Dark Action Bar

Choose the base theme to use for the application

Fig. 2.2 | **New Android Application** dialog.

2.3.3 New Android Application Dialog

In the **New Android Application** dialog's first step (Fig. 2.2), specify the following information, then click **Next >**:

1. **Application Name:** field—Your app's name. Enter `Welcome` in this field.
2. **Project Name:** field—The project's name, which is displayed in the project's *root node* in the IDE's **Package Explorer** tab. By default, the IDE sets this to the app name *without spaces* and with each word capitalized—for an app named `Address Book`, the project name would be `AddressBook`. If you prefer to use a different name, enter it in the **Project name:** field.
3. **Package Name:** field—The Java package name for your app's source code. Android and the Google Play store use this as the app's *unique identifier*, which must remain the same through *all* versions of your app. The package name normally begins with your company's or institution's domain name *in reverse*—ours is `deitel.com`, so we begin our package names with `com.deitel`. Typically, this is followed by the app's name. By convention, package names use only lowercase letters. The IDE specifies a package name that begins with `com.example` by default—this is for learning purposes *only* and must be changed if you intend to distribute your app.
4. **Minimum Required SDK:** field—The *minimum Android API level* that's required to run your app. This allows your app to execute on devices at that API level and *higher*. We use the API level 18, which corresponds to Android 4.3—the lower of the two versions we use in this book. Figure 2.3 shows the Android SDK versions and API levels. Other versions of the SDK are now *deprecated* and should *not* be used. The percentage of Android devices running each platform version is shown at:

```
http://developer.android.com/about/dashboards/index.html
```

SDK version	API level	SDK version	API level	SDK version	API level
4.4	19	4.0.3–4.0.4	15	2.2	8
4.3	18	4.0.1	14	2.1	7
4.2.x	17	3.2	13	1.6	4
4.1.x	16	2.3.3–2.3.7	10		

Fig. 2.3 | Android SDK versions and API levels. (`http://developer.android.com/about/dashboards/index.html`)

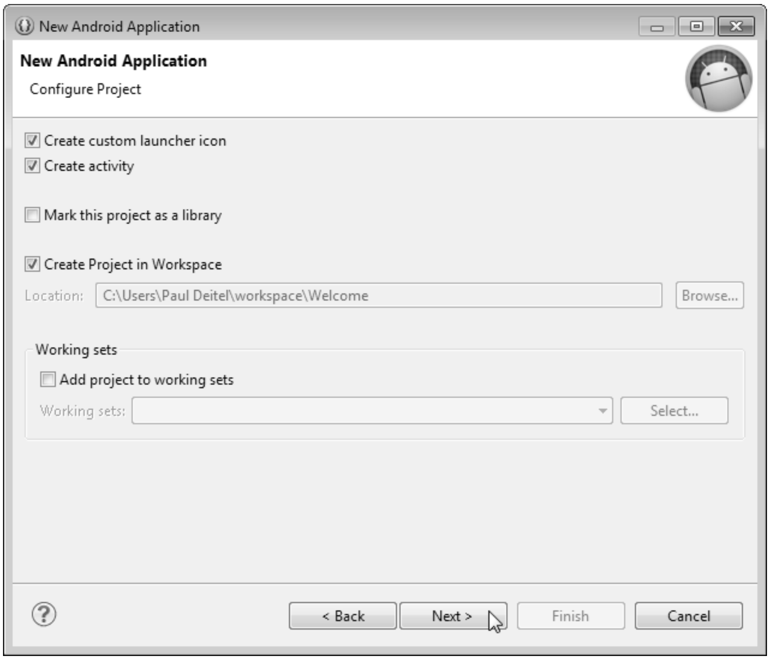
5. **Target SDK:** field—The *preferred* API level. We use level 19 (Android 4.4) for this book's apps. At the time of this writing, 26% of Android devices still used level 10. When developing apps for distribution, you often want to target as many devices as possible. For example, to target devices with Android 2.3.3 and higher (98% of all Android devices), you'd set the **Minimum Required SDK** to 10. If it's set to an earlier API level than the **Target SDK**, *you must ensure either that your app does not use features from API levels above the* ***Minimum Required SDK*** *or that it can detect the API level on the device and adjust its functionality accordingly.* The *Android Lint* tool that the IDE runs in the background points out unsupported features that you use.

6. **Compile With:** field—The version of the API used when compiling your app. Normally this is the same as the **Target SDK**, but it could be an earlier version that supports all the APIs used in your app.
7. **Theme:** field—Your app's default Android *theme*, which gives the app a look-and-feel that's consistent with Android. There are three themes you can choose from—*Holo Light*, *Holo Dark* and *Holo Light with Dark Action Bars* (the default specified by the IDE). When designing a GUI, you can choose from many variations of the Holo Light and Holo Dark themes. For this chapter we'll use the default theme, and we'll discuss themes in more detail in subsequent chapters. For more information about each theme and to see sample screen captures, visit

```
http://developer.android.com/design/style/themes.html
```

2.3.4 Configure Project Step

In the **New Android Application** dialog's **Configure Project** step (Fig. 2.4), leave the default settings as shown and click **Next >**. These settings allow you in subsequent steps to specify your app's icon and configure your app's `Activity`—a class that controls the app's execution.


New Android Application

New Android Application
Configure Project

☑ Create custom launcher icon
☑ Create activity

☐ Mark this project as a library

☑ Create Project in Workspace
Location: C:\Users\Paul Deitel\workspace\Welcome

Working sets
☐ Add project to working sets
Working sets:

Fig. 2.4 | **New Android Application** dialog—**New Android Application** step 2.

2.3.5 Configure Launcher Icon Step

When your app is installed on a device, its icon and name appear with all other installed apps in the *launcher*, which you can access via the icon on your device's home screen. Android runs on a wide variety of devices that have different screen sizes and resolutions.

To ensure that your images look good on all devices, you should provide several versions of each image your app uses. Android can automatically choose the correct image based on various specifications, such as the screen's resolution (width and height in pixels) or DPI (dots per inch). We discuss these mechanisms starting in Chapter 3. You can find more information about designing for varying screen sizes and resolutions at

```
http://developer.android.com/training/multiscreen/index.html
```

and about icons in general at

```
http://developer.android.com/design/style/iconography.html
```

The **Configure Launcher Icon** step (Fig. 2.5) enables you to configure the app's icon from an existing image, a piece of clip art or text. It takes what you specify and creates versions scaled to 48-by-48, 72-by-72, 96-by-96 and 144-by-144 to support various screen resolutions. For this app, we used an image named `DeitelOrange.png`. To use it, click **Browse...** to the right of the **Image File:** field, navigate to the `images` folder in the book's examples folder, select `DeitelOrange.png` and click **Open**. Previews of the scaled images are shown in the dialog's **Preview** area. These images will be placed into appropriate folders in the app's project. Images do not always scale well. For apps that you intend to place in the Google Play store, you might want to have an artist design icons for the appropriate resolutions. In Chapter 9, we discuss submitting apps to the Google Play store and list several companies that offer free and fee-based icon design services. Click **Next >** to continue to the **Create Activity** step.

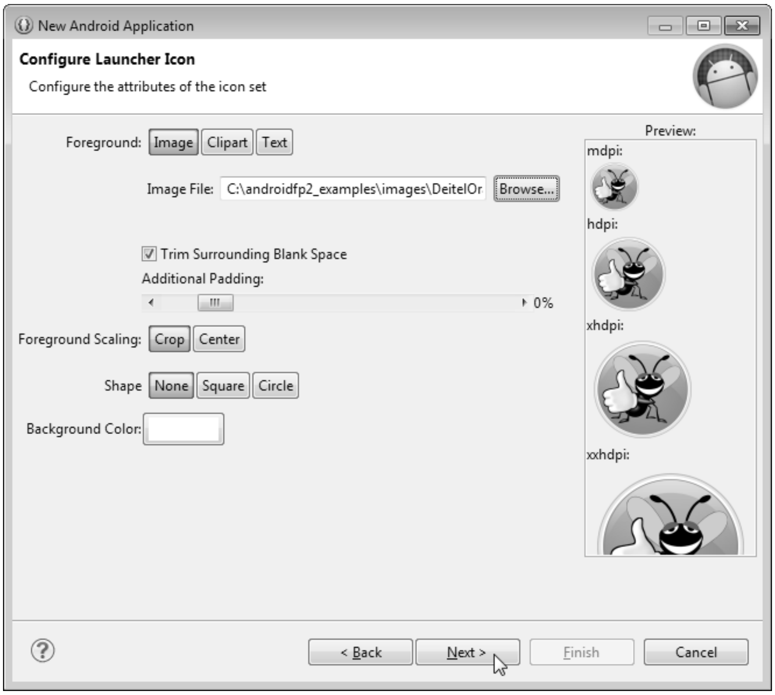
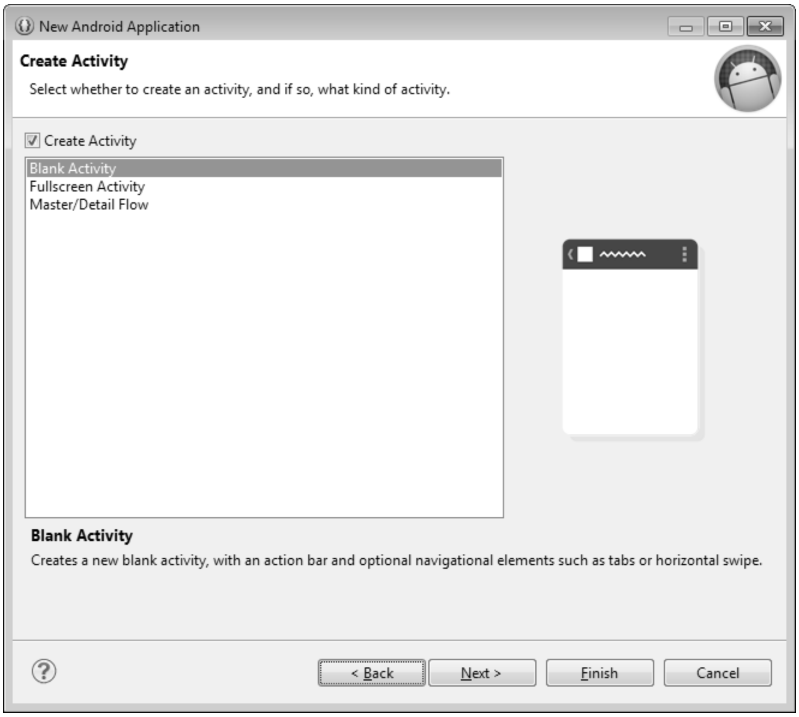

New Android Application
Configure Launcher Icon
Configure the attributes of the icon set
Foreground: Image Clipart Text
Image File: C:\androidfp2_examples\images\DeitelOr
Browse...
Trim Surrounding Blank Space
Additional Padding:
0%
Foreground Scaling: Crop Center
Shape None Square Circle
Background Color:
Preview:
mdpi:
hdpi:
xhdpi:
xxhdpi:
< Back
Next >
Finish
Cancel

Fig. 2.5 | **New Android Application** dialog—**Configure Launcher Icon** step.

2.3.6 Create Activity Step

In the **Create Activity** step (Fig. 2.6), you select the template for your app's `Activity`. **Templates** save you time by providing preconfigured starting points for commonly used app designs. Figure 2.7 briefly describes the three templates shown in Fig. 2.6. For this app, select **Blank Activity**, then click **Next >**. We'll use the other templates in later chapters.



New Android Application

Create Activity

Select whether to create an activity, and if so, what kind of activity.

☑ Create Activity

Blank Activity
Fullscreen Activity
Master/Detail Flow

Blank Activity

Creates a new blank activity, with an action bar and optional navigational elements such as tabs or horizontal swipe.

Fig. 2.6 | **New Android Application** dialog—**Create Activity** step.

Template	Description
Blank Activity	Used for a *single-screen app* in which you build most of the GUI yourself. Provides an *action bar* at the top of the app that displays the app's name and can display controls that enable a user to interact with the app.
Fullscreen Activity	Used for a *single-screen app* (similar to **Blank Activity**) that occupies the entire screen, but can toggle visibility of the device's status bar and the app's action bar.
Master/Detail Flow	Used for an app that displays a *master list* of items from which a user can choose one item to see its *details*—similar to the built-in **Email** and **People** apps. For tablets, the master list and details are shown side-by-side on the same screen. For phones, the master list is shown on one screen, and selecting an item displays the item's details in a separate screen.

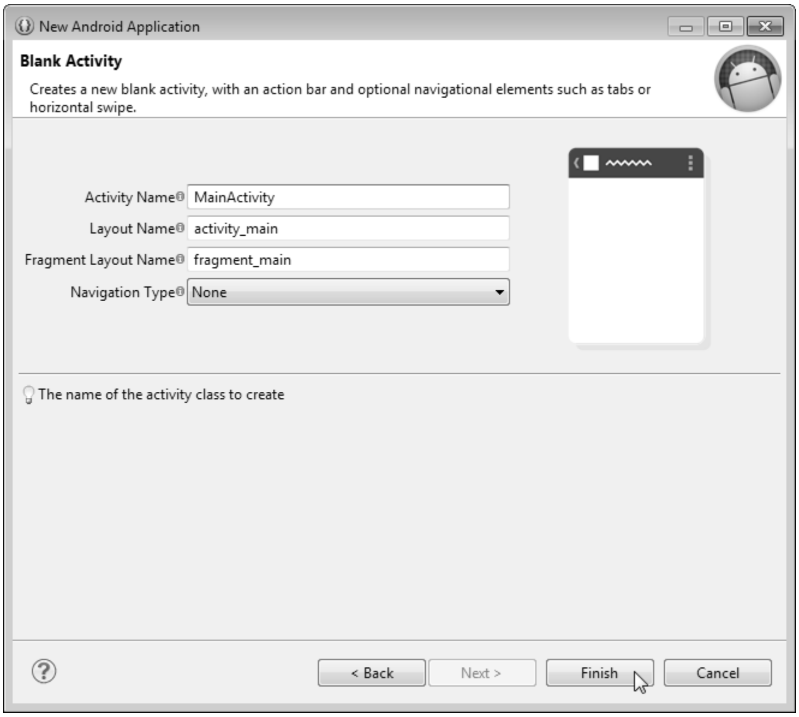
Fig. 2.7 | `Activity` templates.

2.3.7 Blank Activity Step

This step depends on the template selected in the previous step. For the **Blank Activity** template, this step allows you to specify:

- **Activity Name**—`MainActivity` is the default name provided by the IDE. This is the name of a subclass of `Activity` that controls the app's execution. Starting in Chapter 3, we'll modify this class to implement an app's functionality.
- **Layout Name**—`activity_main` is the default file name provided by the IDE. This file stores an XML representation of `MainActivity`'s GUI. In this chapter, you'll build the GUI (Section 2.5) using visual techniques.
- **Fragment Layout Name**—`fragment_main` is the default file name provided by the IDE. An activity's GUI typically contains one or more fragments that describe portions of the activity's GUI. In the default app template, `activity_main` displays the GUI described by `fragment_main`. We discuss fragments in detail starting in Chapter 5. Until then we'll simply ignore the `fragment_main` file.
- **Navigation Type**—`None` is the default specified by the IDE. The **Welcome** app does not provide any functionality. In an app that supports user interactions, you can select an appropriate **Navigation Type** to enable the user to browse through your app's content. We'll discuss navigation options in more detail in later apps.

Click **Finish** to create the project.


New Android Application

Blank Activity

Creates a new blank activity, with an action bar and optional navigational elements such as tabs or horizontal swipe.

Activity Name	MainActivity
Layout Name	activity_main
Fragment Layout Name	fragment_main
Navigation Type	None

The name of the activity class to create

Fig. 2.8 | **New Android Application** dialog—**Blank Activity** step.

2.4 Android Developer Tools Window

After creating the project, the IDE opens `MainActivity.java` and `fragment_main.xml`. Close these, then open `activity_main.xml` from the project's `res/layout` folder, so the IDE appears as shown in Fig. 2.9. The IDE shows the **Graphical Layout** editor so you can begin designing your app's GUI. In this chapter, we discuss only the IDE features we need to build the **Welcome** app. We'll introduce many more IDE features throughout the book.

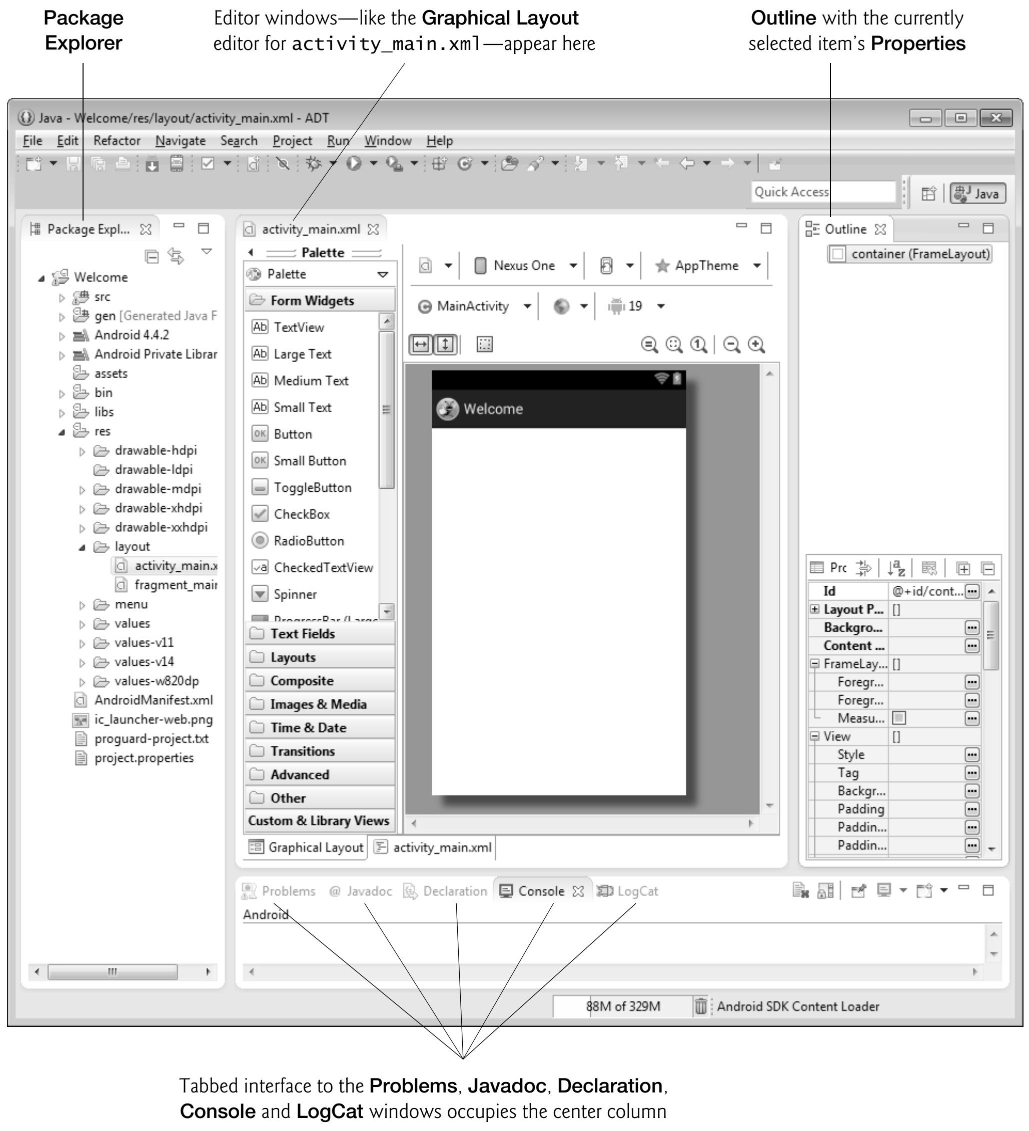


Fig. 2.9 | `Welcome` project open in the Android Developer Tools.

2.4.1 Package Explorer Window

The **Package Explorer** window provides access to all of the project's files. Figure 2.10 shows the **Welcome** app project in the **Package Explorer** window. The **Welcome** node represents the project. You can have many projects open in the IDE at once—each will have its own *top-level node*. Within a project's node, the contents are organized into folders and files. In this chapter, you'll use only files located in the `res` folder, which we discuss in Section 2.4.4—we'll discuss the other folders as we use them in later chapters.

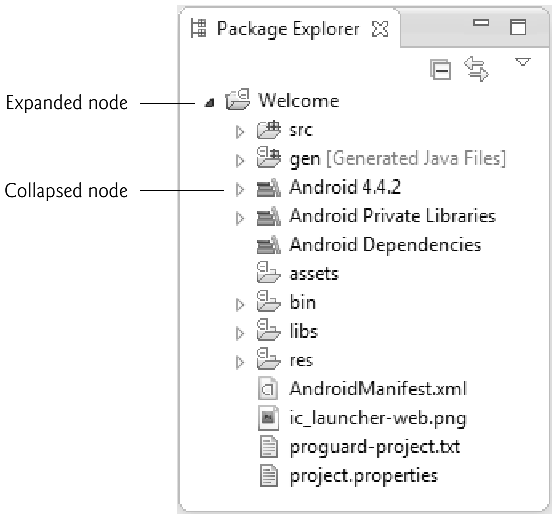


Fig. 2.10 | **Package Explorer** window.

2.4.2 Editor Windows

To the right of the **Package Explorer** in Fig. 2.9 is the **Graphical Layout** editor window. When you double click a file in the **Package Explorer**, its contents are displayed in an appropriate editor window, depending on the file's type. For a Java file, the Java source-code editor is displayed. For an XML file that represents a GUI (such as `activity_main.xml`), the **Graphical Layout** editor is displayed.

2.4.3 Outline Window

The **Outline** is displayed at the right side of the IDE (Fig. 2.9). This window shows information related to the file that's currently being edited. For a GUI, this window shows all the elements that compose the GUI. For a Java class, it shows the class's name and its methods and fields.

2.4.4 App Resource Files

Layout files like `activity_main.xml` (in the project's `res/layout` folder) are considered app *resources* and are stored in the project's **`res`** folder. Within that folder are subfolders for different resource types. The ones we use in this app are shown in Fig. 2.11, and the others (`menu`, `animator`, `anim`, `color`, `raw` and `xml`) are discussed as we need them through the book.

Resource subfolder	Description
`drawable`	Folder names that begin with `drawable` typically contain images. These folders may also contain XML files representing shapes and other types of drawables (such as the images that represent a button's *unpressed* and *pressed* states).
`layout`	Folder names that begin with `layout` contain XML files that describe GUIs, such as the `activity_main.xml` file.
`values`	Folder names that begin with `values` contain XML files that specify values for *arrays* (`arrays.xml`), *colors* (`colors.xml`), *dimensions* (`dimen.xml`; values such as widths, heights and font sizes), *strings* (`strings.xml`) and *styles* (`styles.xml`). These file names are used by convention but are *not* required—actually, you can place all resources of these types in *one* file. It's considered best practice to define the data from hard-coded arrays, colors, sizes, strings and styles as *resources* so they can be modified easily without changing the app's Java code. For example, if a *dimension resource* is referenced from many locations in your code, you can change the resource file once, rather than locating all occurrences of a hard-coded dimension value in your app's Java source files.

Fig. 2.11 | Subfolders of the project's `res` folder that are used in this chapter.

2.4.5 Graphical Layout Editor

When you first create a project, the IDE opens the app's `fragment_main.xml` file in the **Graphical Layout** editor. If you have not already done so, close this file then double click `activity_main.xml` file in your app's `res/layout` folder to open it in the **Graphical Layout** editor (Fig. 2.12).

Selecting the Screen Type for GUI Design

Android devices can run on many types of devices. In this chapter, you'll design an Android phone GUI. As we mentioned in the Before You Begin section, we use an AVD that emulates the Google Nexus 4 phone for this purpose. The **Graphical Layout** editor comes with many device configurations that represent various screen sizes and resolutions that you can use to design your GUI. For this chapter, we use the predefined **Nexus 4**, which we selected in the screen-type drop-down list in Fig. 2.12. This does not mean that the app can execute only on a Nexus 4 device—it simply means that the design is for devices similar in screen size and resolution to the Nexus 4. In later chapters, you'll learn how to design your GUIs to scale appropriately for a wide range of devices.

2.4.6 The Default GUI

The default GUI (Fig. 2.12) for a **Blank Page** app consists of a `FrameLayout` (named `container`) with a light background (specified by the theme we chose when creating the proj-

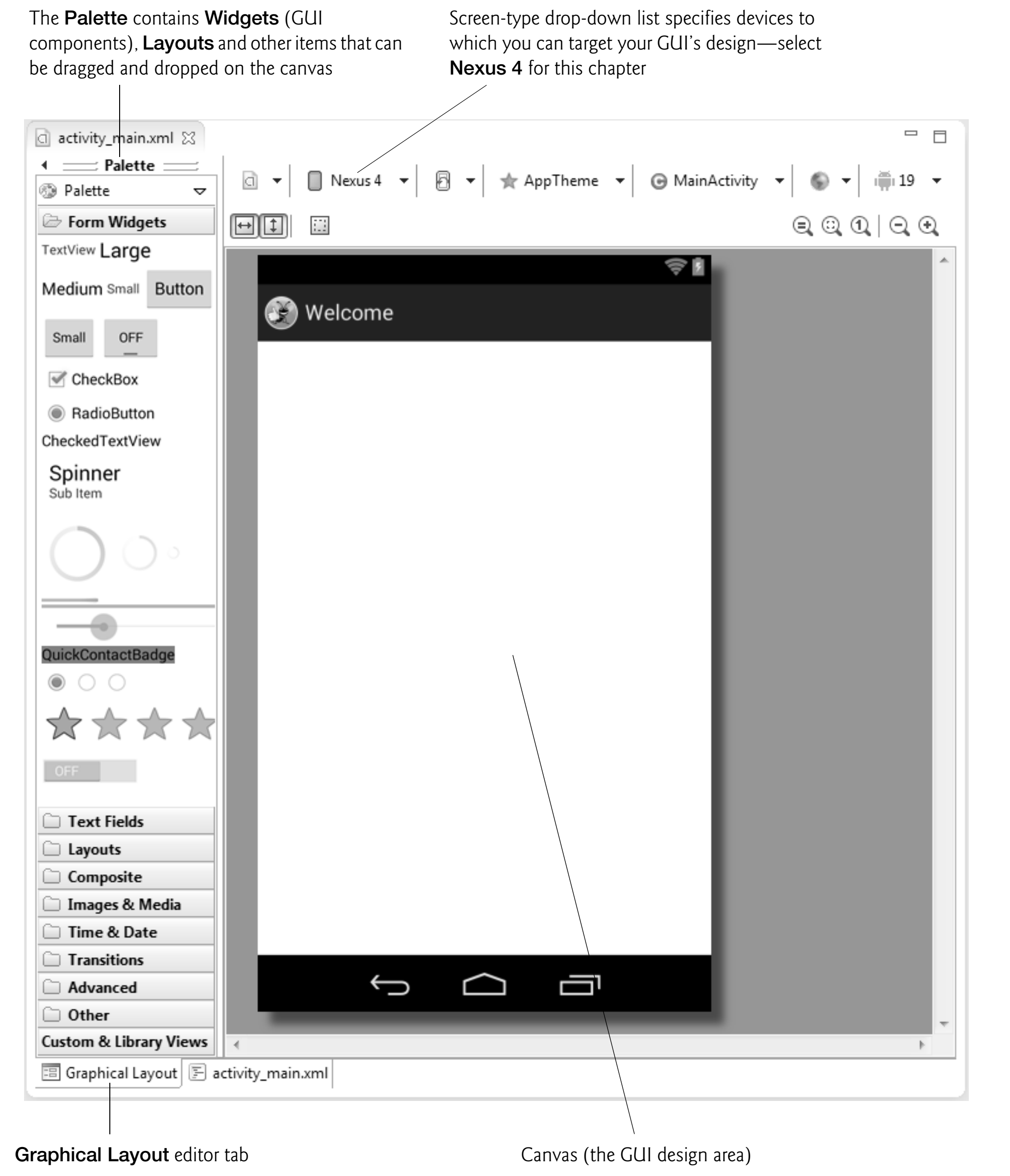


Fig. 2.12 | **Graphical Layout** editor view of the app's default GUI.

ect). A **`FrameLayout`** is designed to display only one GUI component—typically a layout that contains many other GUI components. In this app, you'll use a **`RelativeLayout`** that arranges GUI components *relative to one another* or *relative to the layout itself*—for example, you can specify that a GUI component should appear *below* another GUI component and *centered horizontally* within the `RelativeLayout`. A **`TextView`** displays text. We'll say more about each of these in Section 2.5.

2.5 Building the App's GUI with the Graphical Layout Editor

The IDE's **Graphical Layout** editor allows you to build a GUI by dragging and dropping components—such as `TextView`s, `ImageView`s and `Button`s—onto a design area. By default, the GUI layout for a **Blank App**'s `MainActivity` is stored in the XML file **`activity_main.xml`**, located in the project's `res` folder in the `layout` subfolder. In this chapter, we'll use the **Graphical Layout** editor and the **Outline** window to build the GUI and will *not* study the generated XML. The Android development tools have improved to the point that, in most cases, you do not need to manipulate the XML markup directly.

2.5.1 Adding Images to the Project

For this app, you'll need to add the Deitel bug image (`bug.png`) and the Android logo image (`android.png`) to the project. These are located with the book's examples in the `images` folder's `Welcome` subfolder. *File names for image resources—and all the other resources you'll learn about in later chapters—must be in all lowercase letters.*

Because Android devices have various *screen sizes*, *resolutions* and *pixel densities* (that is, dots per inch or DPI), you typically provide images in varying resolutions that the operating system chooses based on a device's pixel density. For this reason your project's `res` folder contains several subfolders that begin with the name `drawable`. These folders store images with different pixel densities (Fig. 2.13).

Density	Description
`drawable-ldpi`	*Low density*—approximately 120 dots-per-inch.
`drawable-mdpi`	*Medium density*—approximately 160 dots-per-inch.
`drawable-hdpi`	*High density*—approximately 240 dots-per-inch.
`drawable-xhdpi`	*Extra high density*—approximately 320 dots-per-inch.
`drawable-xxhdpi`	*Extra Extra high density*—approximately 480 dots-per-inch.
`drawable-xxxhdpi`	*Extra Extra Extra high density*—approximately 640 dots-per-inch.

Fig. 2.13 | Android pixel densities.

Images for devices that are similar in pixel density to the Google Nexus 4 phone we use in our phone AVD are placed in the folder `drawable-hdpi`. Images for devices with higher pixel densities (such as those on some phones and tablets) are placed in the `drawable-xhdpi` or `drawable-xxhdpi` folders. Images for the medium- and low-density screens of older Android devices are placed in the folders `drawable-mdpi` and `drawable-ldpi`, respectively.

For this app, we provide only one of each image. If Android cannot find an image in the appropriate `drawable` folder, it scales the version from another `drawable` folder up or down to different densities as necessary.

Perform the following steps to add the images to this project:

1. In the **Package Explorer** window, expand the project's `res` folder.
2. Locate and open `images` folder's `Welcome` subfolder on your file system, then drag the images onto the `res` folder's `drawable-hdpi` subfolder. In the **File Operation**

dialog that appears, ensure that **Copy Files** is selected, then click **OK**. In general, you should use PNG images, but JPG and GIF images are also supported.

These images can now be used in the app.

Look-and-Feel Observation 2.1

Low-resolution images do not scale well. For images to render nicely, a high-pixel-density device needs higher resolution images than a low-pixel-density device.

Look-and-Feel Observation 2.2

For detailed information on supporting multiple screens and screen sizes in Android, visit `http://developer.android.com/guide/practices/screens_support.html`.

2.5.2 Changing the `FrameLayout` to a `RelativeLayout`

When a GUI is displayed in the **Graphical Layout** editor, you can use the **Properties** window at the bottom of the **Outline** window to configure the selected layout's or component's properties without editing the XML directly. To select a layout or component, either select its node in the **Outline** window (Fig. 2.14) or click it in the **Graphical Layout** editor. Selecting specific components is often easier in the **Outline** window.

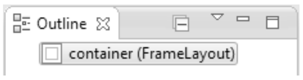


Fig. 2.14 | Hierarchical GUI view in the **Outline** window.

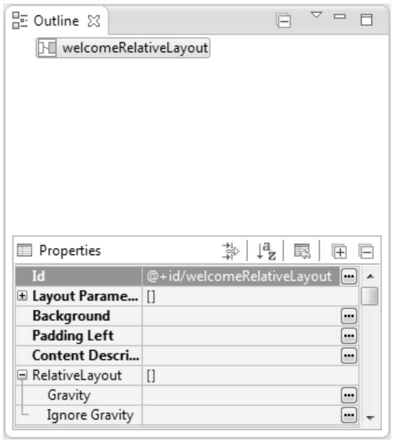
To begin building the GUI, right click the `container FrameLayout` in the **Outline** window, select **Change Layout...**, then select **Relative Layout** and click **OK**.

You should give a relevant name to each layout and component—especially if it will be manipulated programmatically (as we'll do in later apps). This is done via the component's **Id property**—the default **Id** for the `FrameLayout` in `activity_main.xml` is `container`, which you'll change. You can use the **Id** to access and modify a component in a layout and from Java code. As you'll soon see, the **Id** is used to specify the *relative positioning* of components in a `RelativeLayout`. At the top of the **Properties** window set the **Id** value to

```
@+id/welcomeRelativeLayout
```

and press *Enter*. In the **Update References** dialog, click **Yes**, then in the **Rename Resource** dialog, click **OK** to complete the change. The + in the syntax `@+id` indicates that a *new* `id` for referring to that GUI component should be created with the identifier to the right of the forward slash (/). The **Properties** window should now appear as in Fig. 2.15.

In most apps, you should provide some extra space around a layout—known as padding—to separate the layout's components from those in other layouts or from the device's screen edges. Due to recent changes in Google's default app template, this padding is no longer provided in `activity_main.xml`. To add it for this app, scroll to the **Properties** window's **View** subsection. For the **Padding Left** and **Padding Right** properties click the ellipsis (...) button, then select `activity_horizontal_margin` and click **OK**. Repeat this for **Padding Top** and **Padding Bottom** but select `activity_vertical_margin`. We'll discuss padding in more detail in the next chapter.



welcomeRelativeLayout

Id	@+id/welcomeRelativeLayout
⊞ Layout Parame...	[]
Background	
Padding Left	
Content Descri...	
⊟ RelativeLayout	[]
Gravity	
Ignore Gravity	

Fig. 2.15 | **Properties** window after changing the **Id** property of the `RelativeLayout`.

2.5.3 Adding and Configuring a `TextView`

Adding the `TextView` and Setting Its Id Property

To add the `TestView` to the GUI, from the **Palette** at the **Graphical Layout** editor's left side, drag a `TextView` onto the `welcomeRelativeLayout` node in the **Outline** window. By default, the IDE gives the TextView the Id textView1. With the `TextView` selected in the **Outline** window, change its **Id** property to

```
@+id/welcomeTextView
```

Configuring the `TextView`'s Text Property Using a String Resource

According to the Android documentation for application resources

```
http://developer.android.com/guide/topics/resources/index.html
```

it's considered a good practice to place strings, string arrays, images, colors, font sizes, dimensions and other app resources in XML file within the subfolders of the project's `res` folder, so that the resources can be managed separately from your app's Java code. This is known as *externalizing* the resources. For example, if you externalize color values, all components that use the same color can be updated to a new color simply by changing the color value in a central resource file.

If you wish to *localize* your app in several languages, storing the strings *separately* from the app's code allows you to change them easily. In your project's `res` folder, the subfolder `values` contains a `strings.xml` file that's used to store the app's default language strings—English for our apps. To provide localized strings for other languages, you can create separate `values` folders for each language, as we'll demonstrate in Section 2.8.

To set the `TextView`'s **Text property**, create a new string resource in the `strings.xml` file as follows:

1. Ensure that the `welcomeTextView` is selected.
2. Locate its **Text** property in the **Properties** window, then click the ellipsis button to the right of the property's value to display the **Resource Chooser** dialog.

3. In the **Resource Chooser** dialog, click the **New String...** button to display the **Create New Android String** dialog (Fig. 2.16).

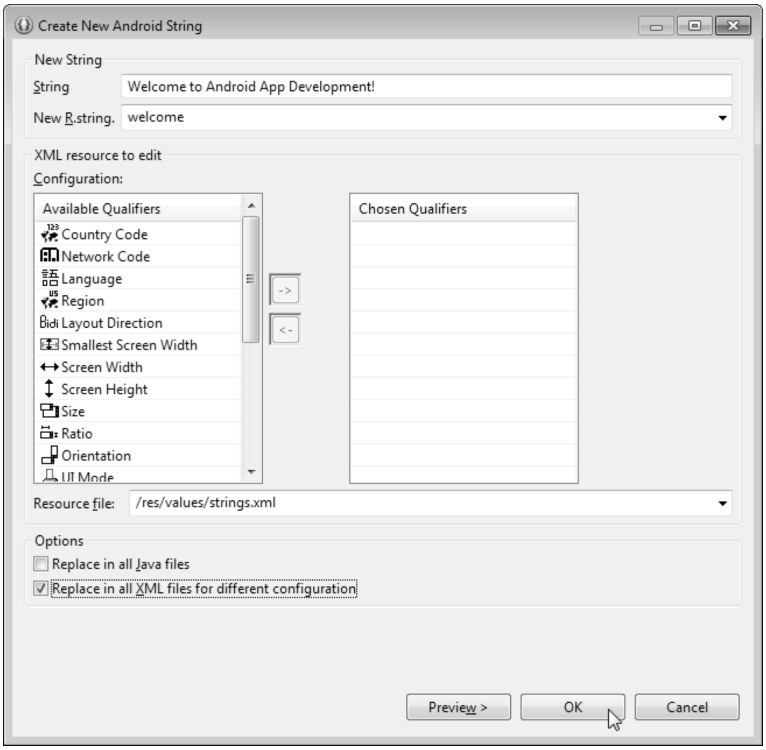
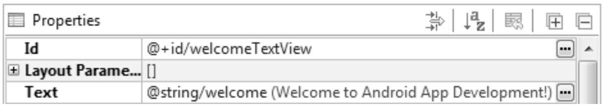


Fig. 2.16 | **Create New Android String** dialog.

4. Fill the **String** and **New R.string** fields as shown in Fig. 2.16, check the **Replace in all XML file for different configurations** checkbox then click **OK** to dismiss the dialog and return to the **Resource Chooser** dialog. The **String** field specifies the text that will be displayed in the `TextView`, and the **R.string** field specifies the string resource's name so that we can reference it in the `TextView`'s **Text** property.

5. The new string resource named `welcome` is automatically selected. Click **OK** in the **Resource Chooser** dialog to use this resource.

In the **Properties** window, the **Text** property should now appear as in Fig. 2.17. The syntax `@string` indicates that a string resource will be selected from the `strings.xml` file (located in the project's `res/values` folder) and `welcome` indicates which string resource to select.



Properties	
Id	@+id/welcomeTextView
⊞ Layout Parame...	[]
Text	@string/welcome (Welcome to Android App Development!)

Fig. 2.17 | **Properties** window after changing the `TextView`'s `Text` property.

Configuring the `TextView`'s **Text Size** *Property—Scaled Pixels and Density-Independent Pixels*

The sizes of GUI components and text can be specified in various measurement units (Fig. 2.18). The documentation for supporting multiple screen sizes

```
http://developer.android.com/guide/practices/screens_support.html
```

recommends that you use *density-independent pixels* for the dimensions of GUI components and other screen elements, and *scale-independent pixels* for font sizes.

Unit	Description
`px`	pixel
`dp` or `dip`	density-independent pixel
`sp`	scale-independent pixel
`in`	inches
`mm`	millimeters

Fig. 2.18 | Measurement units.

Defining your GUIs with **density-independent pixels** enables the Android platform to *scale* the GUI, based on the pixel density of a given device's screen. One *density-independent pixel* is equivalent to one pixel on a 160-dpi screen. On a 240-dpi screen, each density-independent pixel will be scaled by a factor of 240/160 (i.e., 1.5). So, a component that's 100 *density-independent pixels* wide will be scaled to 150 *actual pixels* wide. On a screen with 120 dpi, each density-independent pixel is scaled by a factor of 120/160 (i.e., 0.75). So, the same component that's 100 density-independent pixels wide will be 75 actual pixels wide. ***Scale-independent pixels*** are scaled like density-independent pixels, and they're also scaled by the user's *preferred font size* (as specified in the device's settings).

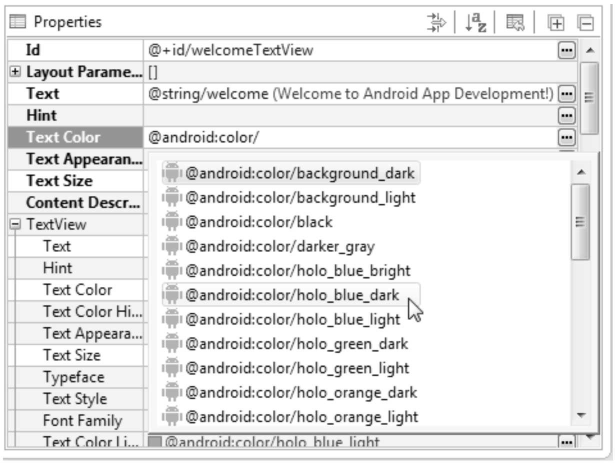
You'll now increase the `TextView`'s font size and add some padding above the `TextView` to separate the text from the edge of the device's screen. To change the font size:

1. Ensure that the `welcomeTextView` is selected.
2. Locate its **Text Size property** in the **Properties** window, then click the ellipsis button to the right of the property's value to display the **Resource Chooser** dialog.
3. In the **Resource Chooser** dialog, click the **New Dimension...** button.
4. In the dialog that appears, specify `welcome_textsize` for the **Name** and `40sp` for the **Value**, then click **OK** to dismiss the dialog and return to the **Resource Chooser** dialog. The letters `sp` in the value `40sp` indicate that this is a *scale-independent pixel* measurement. The letters `dp` in a dimension value (e.g., `10dp`) indicate a *density-independent pixel* measurement.
5. The new dimension resource named `welcome_textsize` is automatically selected. Click **OK** to use this resource.

Configuring Additional `TextView` Properties

Use the **Properties** window to specify the following additional `TextView` properties:

- Set its **Text Color property** to `@android:color/holo_blue_dark`. Android has various predefined color resources. When you type `@android:color/` in the **Text Color** property's value field, a drop-down list of color resources appears (Fig. 2.19). Select `@android:color/holo_blue_dark` from that list to make the text bright blue.



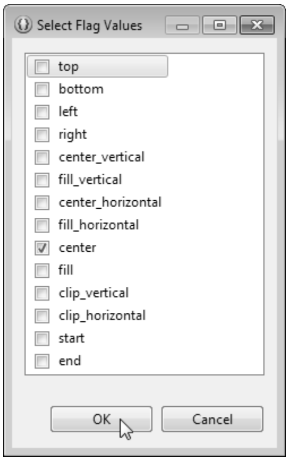
Id	@+id/welcomeTextView
Layout Parame...	[]
Text	@string/welcome (Welcome to Android App Development!)
Hint	
Text Color	@android:color/

Properties
Text Appearan...
Text Size
Content Descr...
TextView
Text
Hint
Text Color
Text Color Hi...
Text Appeara...
Text Size
Typeface
Text Style
Font Family
Text Color Li...
@android:color/background_dark
@android:color/background_light
@android:color/black
@android:color/darker_gray
@android:color/holo_blue_bright
@android:color/holo_blue_dark
@android:color/holo_blue_light
@android:color/holo_green_dark
@android:color/holo_green_light
@android:color/holo_orange_dark
@android:color/holo_orange_light
@android:color/holo_blue_light

Fig. 2.19 | Setting a `TextView`'s **Text Color** property to `@android:color/holo_blue_dark`.

- To center the text in the `TextView` if it wraps to multiple lines, set its **Gravity property** to `center`. To do so, click the **Value** field for this property, then click the ellipsis button to display the **Select Flag Values** dialog with the **Gravity** property's options (Fig. 2.20). Click the **center** checkbox, then click **OK** to set the value.

The **Graphical Layout** editor window should now appear as shown in Fig. 2.21.



Select Flag Values

- ☐ top
- ☐ bottom
- ☐ left
- ☐ right
- ☐ center_vertical
- ☐ fill_vertical
- ☐ center_horizontal
- ☐ fill_horizontal
- ✓ center
- ☐ fill
- ☐ clip_vertical
- ☐ clip_horizontal
- ☐ start
- ☐ end

Fig. 2.20 | Options for the **Gravity** property of an object.

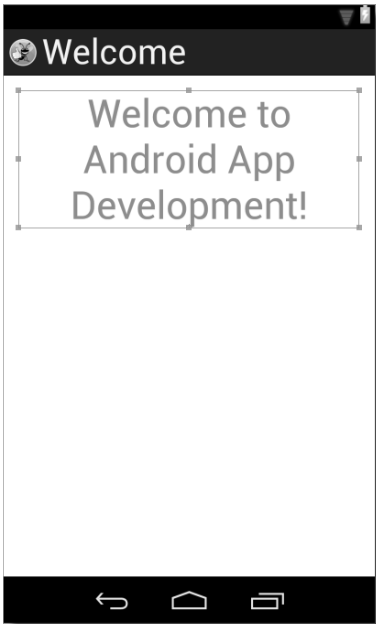


Fig. 2.21 | GUI after completing the `TextView`'s configuration.

2.5.4 Adding `ImageView`s to Display the Images

Next, you'll add two `ImageView`s to the GUI to display the images you added to the project in Section 2.5.1. You'll do this by dragging the `ImageView`s from the **Palette**'s **Images & Media** section onto the GUI below the `TextView`. To do so, perform the following steps:

1. Expand the **Palette**'s **Images & Media** category, then drag an `ImageView` onto the canvas as shown in Fig. 2.22. The new `ImageView` appears below the `welcome-TextView` node. When you drag a component onto the canvas area, the **Graphical Layout** editor displays *green rule markers* and a tooltip appears. The rule markers help you position components in the GUI. The tooltip displays how the GUI component will be configured if you drop it at the current mouse position. The tooltip in Fig. 2.22 indicates that the `ImageView` will be *centered horizontally* in the parent layout (also indicated by the dashed rule marker that extends from the top to the bottom of the GUI) and will be placed below the `welcomeTextView` component (also indicated by the dashed rule marker with an arrowhead).
2. When you drop the `ImageView`, the **Resource Chooser** dialog (Fig. 2.23) appears so that you can choose the image resource to display. For every image you place in a `drawable` folder, the IDE generates a resource ID (i.e., a resource name) that you can use to reference that image in your GUI design and in code. The resource ID is the image's file name without the extension—for `android.png`, the resource ID is `android`. Select `android` and click **OK** to display the droid image. When you add a new component to the GUI, it's automatically selected and its properties are displayed in the **Properties** window.

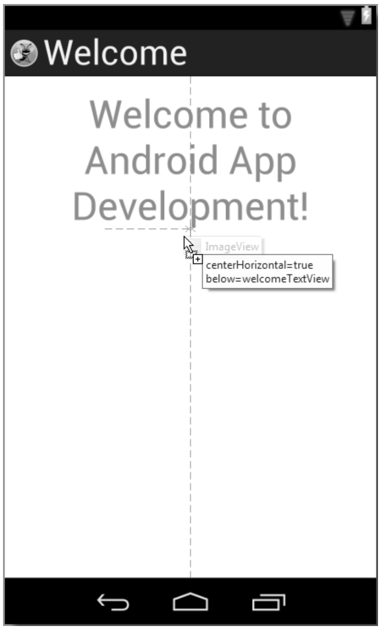


Fig. 2.22 | Dragging and dropping an `ImageView` onto the GUI.

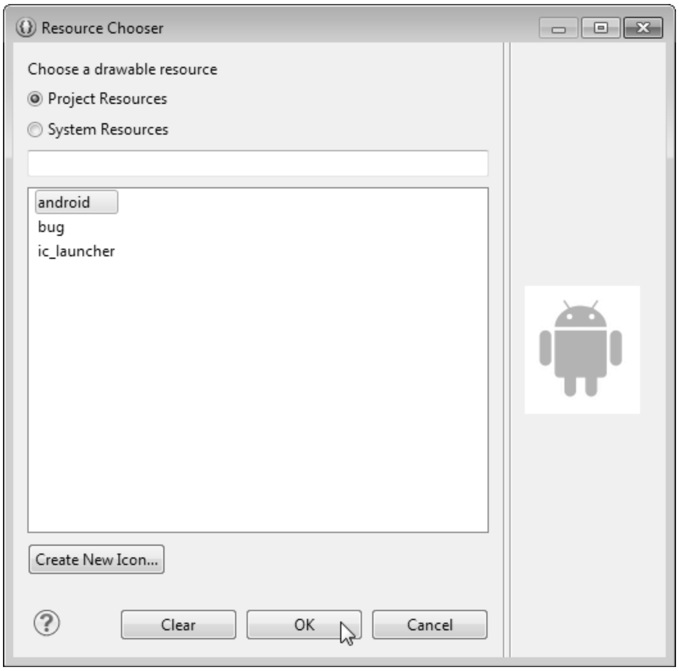
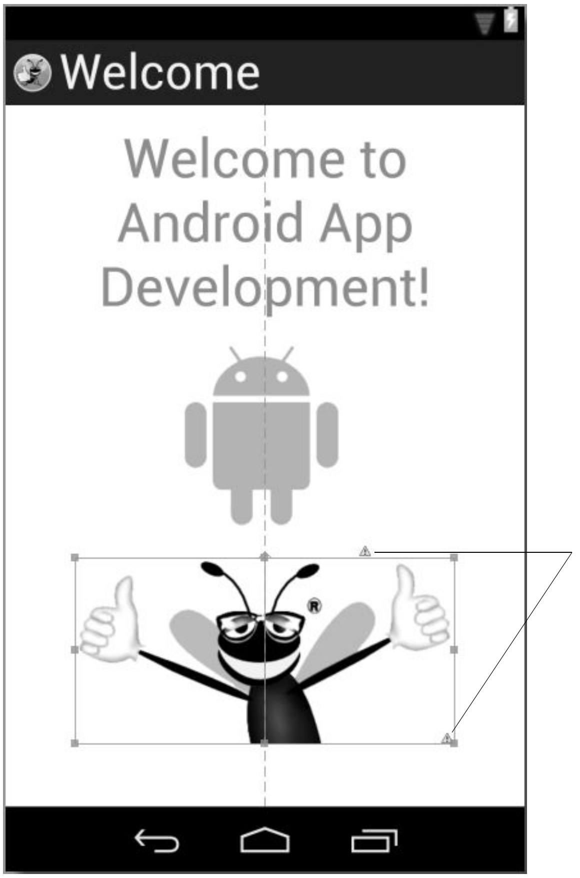


Fig. 2.23 | Selecting the `android` image resource from the **Resource Chooser** dialog.

3. The IDE sets the new `ImageView`'s **Id** property to `@+id/imageView1` by default. Change this to `@+id/droidImageView`. An **Update References?** dialog appears to confirm the *renaming* operation. Click **Yes**. Next, a **Rename Resource** dialog appears to show you all the changes that will be made. Click **OK** to complete the renaming operation.
4. Repeat Steps 1–3 above to create the `bugImageView`. For this component, drag the `ImageView` below the `droidImageView`, select the `bug` image resource from the **Resource Chooser** dialog and set the **Id** property to `@+id/bugImageView` in the **Properties** window, then save the file.

The GUI should now appear as shown in Fig. 2.24.



These warning symbols indicate that an `ImageView` property for visually impaired users is not set for these two images. (We correct this in Section 2.7.)

Fig. 2.24 | Completed GUI design.

2.6 Running the **Welcome** App

To run the app in an *Android Virtual Device (AVD)* for a phone, perform the steps shown in Section 1.9.1. Figure 2.25 shows the running app in the Nexus 4 AVD that you configured in the Before You Begin section. The app is shown in *portrait* orientation, where the device's height is greater than its width. Though you can rotate your device or AVD to *landscape orientation* (where the width is greater than the height), this app's GUI was not designed for that orientation. In the next chapter, you'll learn how to restrict an app's orientation and in subsequent chapters, you'll learn how to create more dynamic GUIs that can handle both orientations.

If you'd like, you can follow the steps in Section 1.9.3 to run the app on an Android device. Though this app will run on an Android tablet AVD or a tablet device, the app's GUI will occupy only a small part of a tablet's screen. Typically, for apps that run on both phones and tablets, you'll also provide a tablet layout that makes better use of the screen's available space, as we'll demonstrate in later chapters.

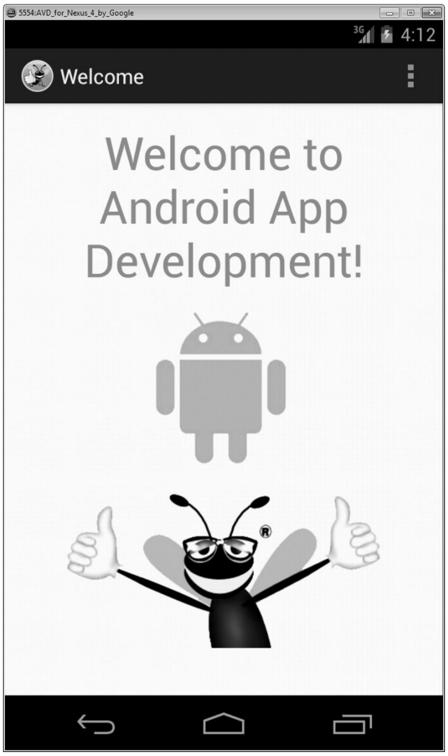

5554:AVD_for_Nexus_4_by_Google
4:12
Welcome
Welcome to Android App Development!

Fig. 2.25 | **Welcome** app running in an AVD.

2.7 Making Your App Accessible

Android contains *accessibility* features to help people with various disabilities use their devices. For people with visual disabilities, Android's **TalkBack** can speak screen text or text that you provide (when designing your GUI or programmatically) to help the user understand the purpose of a GUI component. Android also provides **Explore by Touch**, which enables the user to hear TalkBack speak what's on the screen where the user touches.

When TalkBack is enabled and the user touches an accessible GUI component, TalkBack speaks the component's accessibility text and vibrates the device to provide feedback to users who have trouble hearing. All standard Android GUI components support accessibility. For those that display text, TalkBack speaks that text by default—e.g., when the user touches a `TextView`, TalkBack speaks the `TextView`'s text. You enable TalkBack in the **Settings** app under **Accessibility**. From that page, you can also enable other Android accessibility features such as a *larger default text size* and the ability to use *gestures that magnify areas of the screen*. Unfortunately, TalkBack is *not* currently supported in AVDs, so you must run this app on a device to hear TalkBack speak the text. When you enable TalkBack, Android gives you the option to step through a tutorial of how to use TalkBack with Explore by Touch.

Enabling TalkBack for the `ImageViews`

In the **Welcome** app, we don't need more descriptive text for the `TextView`, because TalkBack will read the `TextView`'s content. For an `ImageView`, however, there is no text for TalkBack to speak unless you provide it. It's considered a best practice in Android to ensure that *every* GUI component can be used with TalkBack by providing text for the **Content Description property** of any component that does not display text. For that reason, the IDE actually warned us that something was wrong with our GUI by displaying small warning (⚠) icons in the **Graphical Layout** editor next to each `ImageView`. These warnings—which are generated by a tool in the IDE known as *Android Lint*—indicate that we did not set the **Content Description** property of each image. The text that you provide should help the user understand the purpose of the component. For an `ImageView`, the text should describe the image.

To add a **Content Description** for each `ImageView` (and eliminate the Android Lint warnings), perform the following steps:

1. Select the `droidImageView` in the **Graphical Layout** editor.
2. In the **Properties** window, click the ellipsis button to the right of the **Content Description** property to open the **Resource Chooser** dialog.
3. Click the **New String...** button to display the **Create New Android String** dialog.
4. In the **String** field specify `"Android logo"` and in the **R.string** field specify `android_logo`, then press **OK**.
5. The new `android_logo` string resource is selected in the **Resource Chooser** dialog, so click **OK** to specify that resource as the value for the `droidImageView`'s **Content Description** property.
6. Repeat the preceding steps for the `bugImageView`, but in the **Create New Android String** dialog, specify `"Deitel double-thumbs-up bug logo"` for the **String** field and `"deitel_logo"` for the **R.string** field. Save the file.

As you set each `ImageView`'s **Content Description**, the warning icon (⚠) for that `ImageView` in the **Graphical Layout** editor is removed.

Testing the App with TalkBack Enabled

Run this app on a device with TalkBack enabled, then touch the `TextView` and each `ImageView` to hear TalkBack speak the corresponding text.

Learning More About Accessibility

Some apps dynamically generate GUI components in response to user interactions. For such GUI components, you can programmatically set the accessibility text. The following Android developer documentation pages provide more information about Android's accessibility features and a checklist to follow when developing accessible apps:

```
http://developer.android.com/design/patterns/accessibility.html
http://developer.android.com/guide/topics/ui/accessibility/index.html
http://developer.android.com/guide/topics/ui/accessibility/
   checklist.html
```

2.8 Internationalizing Your App

As you know, Android devices are used worldwide. To reach the largest possible audience, you should consider customizing your apps for various locales and spoken languages—this is known as **internationalization**. For example, if you intend to offer your app in France, you should translate its resources (e.g., text, audio files) into French. You might also choose to use different colors, graphics and sounds based on the *locale*. For each locale, you'll have a separate, customized set of resources. When the user launches the app, Android automatically finds and loads the resources that match the device's locale settings.

Localization

A key benefit of defining your string values as string resources (as we did in this app) is that you can easily *localize* your app by creating additional XML resource files for those string resources in other languages. In each file, you use the same string-resource names, but provide the *translated* string. Android can then choose the appropriate resource file based on the device user's preferred language.

Naming the Folders for Localized Resources

The XML resource files containing localized strings are placed in subfolders of the project's `res` folder. Android uses a special folder-naming scheme to automatically choose the correct localized resources—for example, the folder `values-fr` would contain a `strings.xml` file for French and the folder `values-es` would contain a `strings.xml` file for Spanish. You can also name these folders with region information—`values-en-rUS` would contain a `strings.xml` file for United States English and `values-en-rGB` would contain a `strings.xml` file for United Kingdom English. If localized resources are not provided for a given locale, Android uses the app's *default* resources—that is, those in the `res` folder's `values` subfolder. We discuss these *alternative-resource naming conventions* in more detail in later chapters.

Adding a Localization Folder to the App's Project

Before you can add a localized version of the **Welcome** app's `strings.xml` file that contains Spanish strings, you must add the `values-es` folder to the project. To do so:

1. In the IDE's **Package Explorer** window, right click the project's `res` folder and select **New > Folder** to display the **New Folder** dialog.
2. In the dialog's **Folder name:** field, enter `values-es`, then click **Finish**.

You'd repeat these steps with an appropriately named `values-`*locale* folder for each language you wish to support.

Copying the `strings.xml` File into the `values-es` Folder

Next, you'll copy the `strings.xml` file from the `values` folder into the `values-es` folder. To do so:

1. In the IDE's **Package Explorer** window, open the res folder's values subfolder, then right click the `strings.xml` file and select **Copy** to copy the file.
2. Next, right click the `values-es` folder, then select **Paste** to place the copy of `strings.xml` in the folder.

Localizing the Strings

In this app, the GUI contains one `TextView` that displays a string and two content-description strings for the `ImageViews`. All of these strings were defined as string resources in the `strings.xml` file. You can now translate the strings in the new version of the `strings.xml` file. App-development companies often have translators on staff or hire other companies to perform translations. In fact, in the Google Play Developer Console—which you use to publish your apps in the Google Play store—you can find translation-services companies. For more information on the Google Play Developer Console, see Chapter 9 and

```
developer.android.com/distribute/googleplay/publish/index.html
```

For this app, you'll replace the strings

```
"Welcome to Android App Development!"
"Android logo"
"Deitel double-thumbs-up bug logo"
```

with the Spanish strings

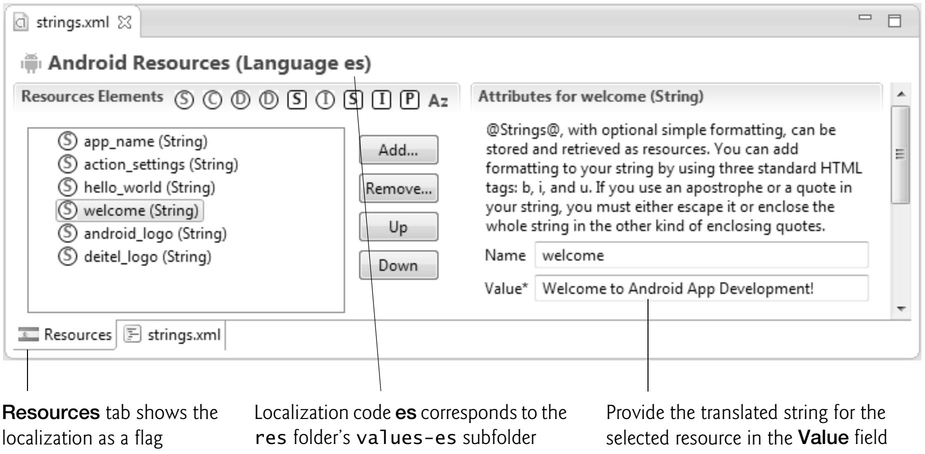
```
"¡Bienvenido al Desarrollo de App Android!"
"Logo de Android"
"El logo de Deitel que tiene el insecto con dedos pulgares
   hacia arriba"
```

To do so:

1. In the IDE's **Package Explorer** window, double click the `strings.xml` file in the `values-es` folder to display the **Android Resources** editor, then select the `welcome` string resource (Fig. 2.26).
2. In the **Value** field, replace the English string `"Welcome to Android App Development!"` with the Spanish string `"¡Bienvenido al Desarrollo de App Android!"`. If you cannot type special Spanish characters and symbols on your keyboard, you can copy the Spanish strings from our `res/values-es/strings.xml` file in the final version of the **Welcome** app (located in the `WelcomeInternationalized` folder with the chapter's examples). To paste the Spanish string into the **Value** field, select the English string, then right click it and select **Paste**.
3. Next, select the `android_logo` resource and change its **Value** to `"Logo de Android"`.
4. Finally, select the `deitel_logo` resource and change its **Value** to `"El logo de Deitel que tiene el insecto con dedos pulgares hacia arriba"`.
5. Delete the `app_name`, `action_settings` and `hello_world` string resources by selecting one at a time and clicking the **Remove...** button. You'll be asked to con-

firm each delete operation. These three resources were placed in the default `strings.xml` file when you created the app's project. Only the `app_name` string resource is used in this project. We'll explain why we deleted it momentarily.

6. Save the `strings.xml` file by selecting **File > Save** or clicking the toolbar icon.



@Strings@, with optional simple formatting, can be stored and retrieved as resources. You can add formatting to your string by using three standard HTML tags: b, i, and u. If you use an apostrophe or a quote in your string, you must either escape it or enclose the whole string in the other kind of enclosing quotes.

Name: welcome

Value*: Welcome to Android App Development!

Resources tab shows the localization as a flag

Localization code **es** corresponds to the `res` folder's `values-es` subfolder

Provide the translated string for the selected resource in the **Value** field

Fig. 2.26 | **Android Resources** editor with the welcome string resource selected.

Testing the App in Spanish

To test the app in Spanish, you must change the language settings in the Android emulator (or on your device). To do so:

1. Touch the home () icon on the emulator or on your device.
2. Touch the launcher () icon, then locate and touch the **Settings** app () icon.
3. In the **Settings** app, scroll to the **PERSONAL** section, then touch **Language & input**.
4. Touch **Language** (the first item in the list), then select **Español** (**España**) from the list of languages.

The emulator or device changes its language setting to Spanish and returns to the **Language & input** settings, which are now displayed in Spanish.

Next, run the **Welcome** app from the IDE, which installs and runs the internationalized version. Figure 2.27 shows the app running in Spanish. When the app begins executing, Android checks the AVD's (or device's) language settings, determines that the AVD (or device) is set to Spanish and uses the `welcome`, `android_logo` and `deitel_logo` string resources defined in `res/values-es/strings.xml` in the running app. Notice, however, that the app's name still appears in *English* in the action bar at the top of the app. This is because we did *not* provide a localized version of the `app_name` string resource in the `res/values-es/strings.xml` file. Recall that when Android cannot find a localized version of a string resource, it uses the default version in the `res/values/strings.xml` file.

Common Programming Error 2.1
Modifying the names of resources can lead to runtime errors. Android uses the default resource names when loading localized resources. When you create a localized resource file, be sure to modify only the values of the resources, not their names.

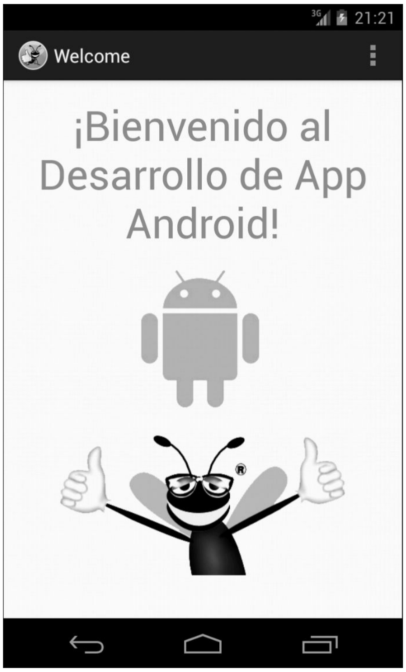


Fig. 2.27 | **Welcome** app running in Spanish in the Nexus 4 AVD.

Returning the AVD (or Device) to English
To return your AVD (or Device) to English:

1. Touch the home (⌂) icon on the emulator or on your device.
2. Touch the launcher (⊞) icon, then locate and touch the **Settings** app (☰) icon—the app is now called **Ajustes** in Spanish.
3. Touch the item **Idioma y entrada de texto** to access the language settings.
4. Touch the item **Idioma**, then in the list of languages select **English (United States)**.

TalkBack and Localization
TalkBack currently supports English, Spanish, Italian, French and German. If you run the **Welcome** app on a device with Spanish specified as the device's language and TalkBack enabled, TalkBack will speak the app's Spanish strings as you touch each GUI component.

When you first switch your device to Spanish and enable TalkBack, Android will automatically download the Spanish text-to-speech engine. If TalkBack does *not* speak the Spanish strings, then the Spanish text-to-speech engine has not finished downloading and installing yet. In this case, you should try executing the app again later.

Localization Checklist
For more information on localizing your app's resources, be sure to check out the Android *Localization Checklist* at:

```
    developer.android.com/distribute/googleplay/publish/localizing.html
```

2.9 Wrap-Up

In this chapter, you used the Android Developer Tools IDE to build the **Welcome** app that displayed a welcome message and two images without writing any code. You created a simple GUI using the IDE's **Graphical Layout** editor and configured properties of GUI components using the **Properties** window.

The app displayed text in a `TextView` and pictures in `ImageViews`. You modified the `TextView` from the default GUI to display the app's text centered in the GUI, with a larger font size and in one of the standard theme colors. You also used the **Graphical Layout** editor's **Palette** of GUI controls to drag and drop `ImageViews` onto the GUI. Following best practices, you defined all strings and numeric values in resource files in the project's `res` folder.

You learned that Android has accessibility features to help people with various disabilities use their devices. We showed how to enable Android's TalkBack to allow a device to speak screen text or speak text that you provide to help the user understand the purpose and contents of a GUI component. We discussed Android's Explore by Touch feature, which enables the user to touch the screen to hear TalkBack speak what's on the screen near the touch. For the app's `ImageViews`, you provided content descriptions that could be used with TalkBack and Explore by Touch.

Finally, you learned how to use Android's internationalization features to reach the largest possible audience for your apps. You localized the **Welcome** app with Spanish strings for the `TextView`'s text and the `ImageViews`' accessibility strings, then tested the app on an AVD configured for Spanish.

Android development is a combination of GUI design and Java coding. In the next chapter, you'll develop a simple **Tip Calculator** app by using the **Graphical Layout** editor to develop the GUI visually and Java programming to specify the app's behavior.

Self-Review Exercises

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

a) Android and the Google Play Store use ________ as the app's unique identifier, which must remain the same through *all* versions of your app.

b) When designing an Android GUI, you typically want it to be ________ so that it displays properly on various devices.

c) Resource folder names that begin with ________ contain XML files that specify values for *arrays*, *colors*, *dimensions*, *strings* and *styles*.

d) Images for the medium-density screens of older Android devices are placed in the folders ________ and ________, respectively.

e) To run an app in an Android Virtual Device (AVD), right click the app's root node in Eclipse in the ________ window and select **Run As > Android Application**.

f) ________ enables the user to hear TalkBack speak what's on the screen where the user touches.

g) Android uses a special folder-naming scheme to automatically choose the correct localized resources—for example, the folder ________ would contain a `strings.xml` file for French and the folder ________ would contain a `strings.xml` file for Spanish.

2.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) An Android app can be compiled using **SDK** with an API level that is lesser than the **Target SDK**, provided that it supports all the APIs used in your app.

b) The layout `RelativeLayout` arranges components relative to one another or relative to their parent container.

c) Android's Explore by Touch feature enables a device to speak out the entire text on the screen.

d) The resource ID in Android is the resource's (such as layout, image) file name without the extension.

e) For people with visual disabilities, Android's SpeakBack can speak screen text or text that you provide to help the user understand the purpose of a GUI component.

f) It's considered a best practice in Android to ensure that every GUI component can be used with TalkBack by providing text for the **Content Description** property of any component that does not display text.

Answers to Self-Review Exercises

2.1 a) package name. b) scalable. c) values. d) `drawable-mdpi, drawable-ldpi`. e) **Project Explorer**. f) Explore by Touch. g) `values-fr`, `values-es`.

2.2 a) True. b) True. c) False. Explore By Touch enables a device to speak what's on the screen near the touch. d) True. e) False. The feature is named TalkBack. f) True.

Exercises

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

a) The ADT's __________ allows you to build GUIs using drag-and-drop techniques.

b) A __________ arranges GUI components relative to one another or relative to the layout itself.

c) Eclipse IDE's __________ window is used to configure various options, such as the `TextView`'s text, font size and font color.

d) Your project's `res` folder contains three subfolders for images—`drawable-hdpi` (high density), `drawable-mdpi` (medium density) and `drawable-ldpi` (low density). These folders store images with different __________ densities.

e) The documentation for supporting multiple screen sizes recommends that you use density-independent pixels for the dimensions of GUI components and other screen elements and __________ for font sizes.

f) One density-independent pixel is equivalent to one pixel on a screen with 160 dpi (dots per inch). On a screen with 240 dpi, each density-independent pixel will be scaled by a factor of __________.

g) On a screen with 120 dpi, each density-independent pixel is scaled by a factor of __________. So, the same component that's 100 density-independent pixels wide will be 75 actual pixels wide.

2.4 State whether each of the following is *true* or *false*. If *false*, explain why.

a) For images to render nicely, a high-pixel-density device needs lower-resolution images than a low-pixel-density device.

b) It's considered a good practice to "externalize" strings, string arrays, images, colors, font sizes, dimensions and other app resources so that you, or someone else on your team, can manage them separately from your application's code.

c) You can use the **Graphical Layout** editor to create a working Android app without writing any code.

2.5 *(Scrapbooking App)* Find open source images of any four nations of your choice. Create an app in which you arrange these flags in a grid format. Add text that identifies each nation's flag. Recall that image file names must use lowercase letters only.

2.6 *(Scrapbooking App with Accessibility)* Using the techniques you learned in Section 2.7, enhance your solution to Exercise 2.5 to provide strings that can be used with Android's TalkBack accessibility feature. If you have an Android device available to you, test the app on the device with TalkBack enabled.

2.7 *(Scrapbooking App with Internationalization)* Using the techniques you learned in Section 2.8, enhance your solution to Exercise 2.6 to define a set of strings for another spoken language. Use an online translator service, such as `translate.google.com` to translate the strings and place them in the appropriate `strings.xml` resource file. Use the instructions in Section 2.8 to test the app on an AVD (or a device if you have one available to you).

Tip Calculator App

3

Objectives

In this chapter you'll:

- Design a GUI using `LinearLayout`s and a `GridLayout`.
- Use the IDE's **Outline** window to add GUI components to `LinearLayout`s and a `GridLayout`.
- Use `TextView`, `EditText` and `SeekBar` GUI components.
- Use Java object-oriented programming capabilities, including classes, objects, interfaces, anonymous inner classes and inheritance to add functionality to an Android app.
- Programmatically interact with GUI components to change the text that they display.
- Use event handling to respond to user interactions with an `EditText` and a `SeekBar`.
- Specify that the keypad should always be displayed when an app is executing.
- Specify that an app supports only portrait orientation.

Outline

3.1 Introduction

The **Tip Calculator** app (Fig. 3.1(a)) calculates and displays possible tips for a restaurant bill. As you enter each digit of a bill amount by touching the *numeric keypad*, the app calculates and displays the tip amount and total bill (bill amount + tip) for a 15% tip and a custom tip

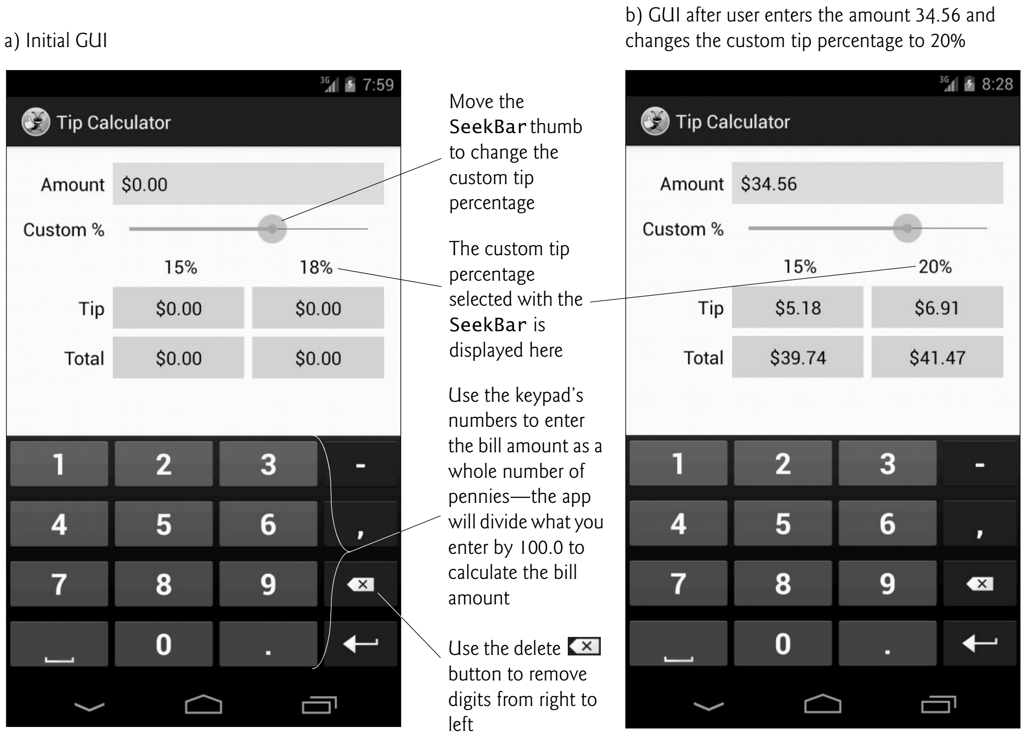

a) Initial GUI
b) GUI after user enters the amount 34.56 and changes the custom tip percentage to 20%
7:59
Tip Calculator
Amount
$0.00
Custom %
15%
18%
Tip
$0.00
$0.00
Total
$0.00
$0.00
8:28
Tip Calculator
Amount
$34.56
Custom %
15%
20%
Tip
$5.18
$6.91
Total
$39.74
$41.47
Move the SeekBar thumb to change the custom tip percentage
The custom tip percentage selected with the SeekBar is displayed here
Use the keypad's numbers to enter the bill amount as a whole number of pennies—the app will divide what you enter by 100.0 to calculate the bill amount
Use the delete button to remove digits from right to left

Fig. 3.1 | Entering the bill total and calculating the tip.

percentage (18% by default). You can specify a custom tip percentage from 0% to 30% by moving the `SeekBar` *thumb*—this updates the custom percentage shown and displays the custom tip and total (Fig. 3.1(b)). We chose 18% as the default custom percentage, because many restaurants in the United States add this tip percentage for parties of six people or more. The keypad in Fig. 3.1 may differ based on your AVD's or device's Android version, or based on whether you've installed and selected a custom keyboard on your device.

You'll begin by test-driving the app—you'll use it to calculate 15% and custom tips. Then we'll overview the technologies you'll use to create the app. You'll build the app's GUI using the Android Developer Tools IDE's **Graphical Layout** editor and the **Outline** window. Finally, we'll present the complete Java code for the app and do a detailed code walkthrough. We provide online an Android Studio version of Sections 3.2 and 3.4 at `http://www.deitel.com/books/AndroidHTP2`.

3.2 Test-Driving the Tip Calculator App

Open and Run the App

Open the Android Developer Tools IDE and import the **Tip Calculator** app project. Perform the following steps:

1. *Launching the Nexus 4 AVD.* For this test-drive, we'll use the Nexus 4 smartphone AVD that you configured in the Before You Begin section. To launch the Nexus 4 AVD, select **Window > Android Virtual Device Manager** to display the **Android Virtual Device Manager** dialog. Select the Nexus 4 AVD and click **Start...**, then click the **Launch** button in the **Launch Options** dialog that appears.
2. *Opening the* **Import** *Dialog.* Select **File > Import...** to open the **Import** dialog.
3. *Importing the* **Tip Calculator** *app's project.* Expand the **General** node, select **Existing Projects into Workspace**, then click **Next >** to proceed to the **Import Projects** step. Ensure that **Select root directory** is selected, then click **Browse...**. In the **Browse For Folder** dialog, locate the `TipCalculator` folder in the book's examples folder, select it and click **OK**. Ensure that **Copy projects into workspace** is *not* selected. Click **Finish** to import the project. It now appears in the **Package Explorer** window.
4. *Launching the* **Tip Calculator** *app.* Right click the `TipCalculator` project in the **Package Explorer** window, then select **Run As > Android Application** to execute **Tip Calculator** in the AVD.

Entering a Bill Total

Using the numeric keypad, enter `34.56`. Just type `3456`—the app will position the cents to the right of the decimal point. If you make a mistake, press the delete (⌫) button to erase one rightmost digit at a time. The `TextView`s under the **15%** and the custom tip percentage (**18%** by default) labels show the tip amount and the total bill for these tip percentages. All the **Tip** and **Total** `TextView`s update each time you enter or delete a digit.

Selecting a Custom Tip Percentage

Use the `Seekbar` to specify a *custom* tip percentage. Drag the `Seekbar`'s *thumb* until the custom percentage reads **20%** (Fig. 3.1(b)). As you drag the thumb, the tip and total for this custom tip percentage update continuously. By default, the `Seekbar` allows you to select values from 0 to 100, but we specified a maximum value of 30 for this app.

3.3 Technologies Overview

This section introduces the IDE features and Android technologies you'll use to build the **Tip Calculator** app. We assume that you're *already* familiar with Java object-oriented programming—if you're not, the appendices contain an introduction to Java. You'll:

- use various Android classes to create objects
- call methods on Android classes and objects
- define and call your own methods
- use inheritance to create a subclass of Android's `Activity` class that defines the **Tip Calculator**'s functionality
- use event handling, anonymous inner classes and interfaces to process the user's GUI interactions

3.3.1 Class `Activity`

Unlike many Java apps, Android apps *don't have a `main` method*. Instead, they have four types of executable components—*activities*, *services*, *content providers* and *broadcast receivers*. In this chapter, we'll discuss activities, which are defined as subclasses of **`Activity`** (package **`android.app`**). Users interact with an `Activity` through *views*—that is, GUI components. Before Android 3.0, a separate `Activity` was typically associated with each screen of an app. As you'll see, starting in Chapter 5, an `Activity` can manage multiple `Fragment`s. On a phone, each `Fragment` typically occupies the entire screen and the `Activity` switches between the `Fragment`s based on user interactions. On a tablet, activities often display multiple `Fragment`s per screen to take better advantage of the larger screen size.

3.3.2 `Activity` Lifecycle Methods

Throughout its life, an `Activity` can be in one of several *states*—*active* (i.e., *running*), *paused* or *stopped*. The `Activity` transitions between these states in response to various *events*:

- An *active* `Activity` is *visible* on the screen and "has the focus"—that is, it's in the *foreground*. This is the `Activity` the user is interacting with.
- A *paused* `Activity` is *visible* on the screen but *does not* have the focus—such as when an alert dialog is displayed.
- A *stopped* activity is *not visible* on the screen and is likely to be killed by the system when its memory is needed. An `Activity` is *stopped* when another `Activity` becomes *active*.

As an `Activity` transitions among these states, the Android runtime calls various `Activity` *lifecycle methods*—all of which are defined in the `Activity` class

```
http://developer.android.com/reference/android/app/Activity.html
```

You'll override the **`onCreate`** method in *every* activity. This method is called by the Android runtime when an `Activity` is *starting*—that is, when its GUI is about to be displayed so that the user can interact with the `Activity`. Other lifecycle methods include `onStart`, `onPause`, `onRestart`, `onResume`, `onStop` and `onDestroy`. We'll discuss *most* of these in later chapters. Each `Activity` lifecycle method you override *must* call the superclass's version; otherwise, an *exception* will occur. This is required because each lifecycle method in superclass `Activity` contains code that must execute in addition to the code you define in your overridden lifecycle methods.

3.3.3 Arranging Views with `LinearLayout` and `GridLayout`

Recall that layouts arrange views in a GUI. A **`LinearLayout`** (package **`android.widget`**) arranges views either *horizontally* (the default) or *vertically* and can size its views proportionally. We'll use this to arrange two `TextView`s horizontally and ensure that each uses half of the available horizontal space.

`GridLayout` (package `android.widget`) was introduced in Android 4.0 as a new layout for arranging views into cells in a rectangular grid. Cells can occupy *multiple* rows and columns, allowing for complex layouts. In many cases, `GridLayout` can be used to replace the older, and sometimes less efficient `TableLayout`, which arranges views into rows and columns where each row is typically defined as a `TableRow` and the number of columns is defined by the `TableRow` containing the most cells. Normally, `GridLayout` requires API level 14 or higher. However, the *Android Support Library* provides alternate versions of `GridLayout` and many other GUI features so that you can use them in older Android versions. For more information on this library and how to use it in your apps, visit:

```
http://developer.android.com/tools/support-library/index.html
```

A `GridLayout` *cannot* specify within a given row that the horizontal space should be allocated *proportionally* between multiple views. For this reason, several rows in this app's GUI will place two `TextView`s in a horizontal `LinearLayout`. This will enable you to place two `TextView`s in the same `GridLayout` cell and divide the cell's space evenly between them. We'll cover more layouts and views in later chapters—for a complete list, visit:

```
http://developer.android.com/reference/android/widget/
   package-summary.html
```

3.3.4 Creating and Customizing the GUI with the Graphical Layout Editor and the Outline and Properties Windows

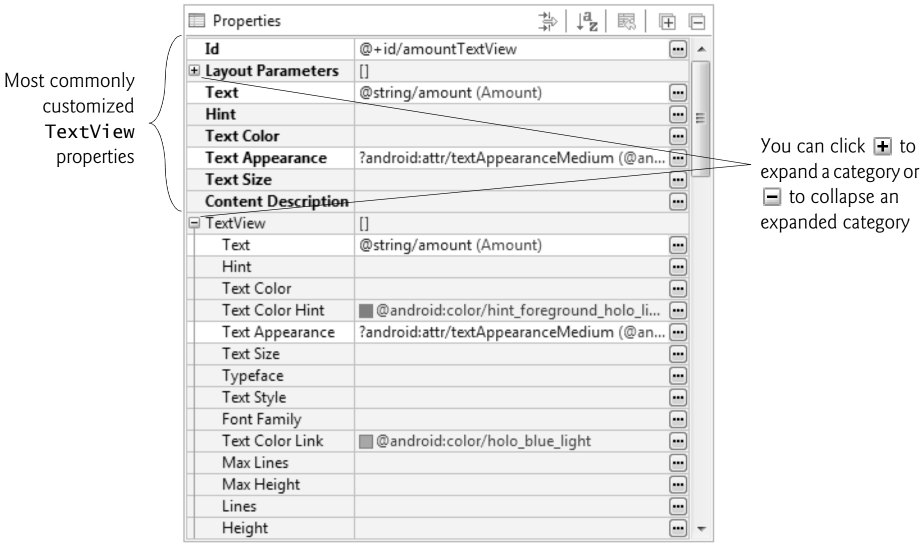
You'll create `TextView`s, an `EditText` and a `SeekBar` using the IDE's **Graphical Layout** editor (that you used in Chapter 2) and **Outline** window, then customize them with the IDE's **Properties** window—which is displayed at the bottom of the **Outline** window when you're editing a GUI in the **Graphical Layout** editor. You'll do this *without* directly manipulating the XML stored in the files of the project's `res` folder.

An **`EditText`**—often called a *text box* or *text field* in other GUI technologies—is a *subclass* of `TextView` (presented in Chapter 2) that can display text *and* accept text input from the user. You'll specify an `EditText` for *numeric* input, allow users to enter only digits and restrict the *maximum* number of digits that can be entered.

A **`SeekBar`**—often called a *slider* in other GUI technologies—represents an integer in the range 0–100 by default and allows the user to select a number in that range by moving the `SeekBar`'s thumb. You'll customize the `SeekBar` so the user can choose a custom tip percentage *only* from the more limited range 0 to 30.

In the **Properties** window, a view's most commonly customized properties typically appear at the top with their names displayed in bold (Fig. 3.2). All of a view's properties are also organized into categories within the **Properties** window. For example, class `TextView` inherits many properties from class `View`, so the **Properties** window displays a **Text-**

View category with `TextView`-specific properties, followed by a **View** category with properties that are inherited from class `View`.


Properties
Most commonly customized TextView properties
You can click ⊞ to expand a category or ⊟ to collapse an expanded category

Id	@+id/amountTextView
⊞ **Layout Parameters**	[]
Text	@string/amount (Amount)
Hint	
Text Color	
Text Appearance	?android:attr/textAppearanceMedium (@an...
Text Size	
Content Description	
⊟ TextView	[]
Text	@string/amount (Amount)
Hint	
Text Color	
Text Color Hint	▨ @android:color/hint_foreground_holo_li...
Text Appearance	?android:attr/textAppearanceMedium (@an...
Text Size	
Typeface	
Text Style	
Font Family	
Text Color Link	▨ @android:color/holo_blue_light
Max Lines	
Max Height	
Lines	
Height	

Fig. 3.2 | **Properties** window showing a `TextView`'s most commonly customized properties.

3.3.5 Formatting Numbers as Locale-Specific Currency and Percentage Strings

You'll use class **`NumberFormat`** (package **`java.text`**) to create *locale-specific* currency and percentage strings—an important part of *internationalization*. You could also add *accessibility* strings and internationalize the app using the techniques you learned in Sections 2.7–2.8, though we did not do so in this app.

3.3.6 Implementing Interface `TextWatcher` for Handling `EditText` Text Changes

You'll use an *anonymous inner class* to implement the **`TextWatcher`** *interface* (from package **`android.text`**) to respond to *events when the user changes the text* in this app's `EditText`. In particular, you'll use method `onTextChanged` to display the currency-formatted bill amount and to calculate the tip and total as the user enters each digit.

3.3.7 Implementing Interface `OnSeekBarChangeListener` for Handling `SeekBar` Thumb Position Changes

You'll implement the `SeekBar.OnSeekBarChangeListener` interface (from package `android.widget`) to respond to the user moving the `SeekBar`'s *thumb*. In particular, you'll use method `onProgressChanged` to display the custom tip percentage and to calculate the custom tip and total as the user moves the `SeekBar`'s thumb.

3.3.8 `AndroidManifest.xml`

The **`AndroidManifest.xml`** file is created by the IDE when you create a new app project. This file contains many of the settings that you specify in the **New Android Application** dialog, such as the app's name, package name, target and minimum SDKs, `Activity` name(s), theme and more. You'll use the IDE's **Android Manifest** editor to add a new setting to the manifest that forces the *soft keyboard* to remain on the screen. You'll also specify that the app supports only *portrait orientation*—that is, the device's longer side is vertical.

3.4 Building the App's GUI

In this section, we'll show the precise steps for building the **Tip Calculator**'s GUI. The GUI will not look like the one shown in Fig. 3.1 until you've completed the steps. As you procede through this section, the number of details presented may seem large, but they're repetitive and you'll get used to them as you use the IDE.

3.4.1 `GridLayout` Introduction

This app uses a **`GridLayout`** (Fig. 3.3) to arrange views into five *rows* and two *columns*. Each cell in a `GridLayout` can be *empty* or can hold one or more *views*, including layouts that *contain* other views. Views can span *multiple* rows or columns, though we did not use that capability in this GUI. You can specify a `GridLayout`'s number of rows and columns in the **Properties** window.

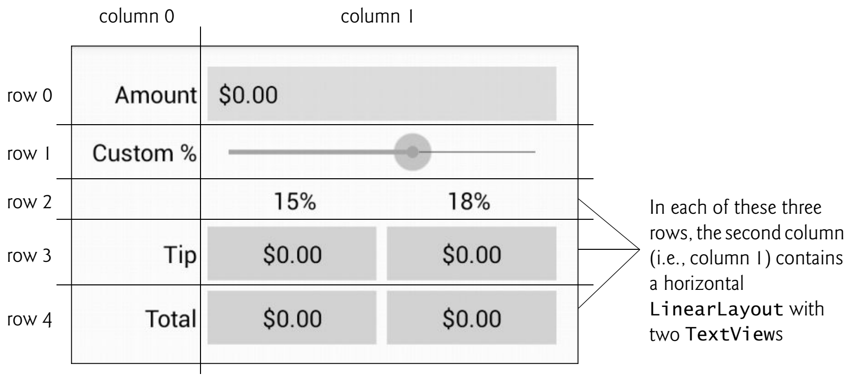

column 0
column 1
row 0
row 1
row 2
row 3
row 4
Amount
$0.00
Custom %
15%
18%
Tip
$0.00
$0.00
Total
$0.00
$0.00
In each of these three rows, the second column (i.e., column 1) contains a horizontal LinearLayout with two TextViews

Fig. 3.3 | **Tip Calculator** GUI's `GridLayout` labeled by its rows and columns.

Each row's *height* is determined by the *tallest* view in that row. Similarly, the *width* of a column is defined by the *widest* view in that column. By default, views are added to a row from left to right. As you'll see, you can specify the exact row and column in which a view is to be placed. We'll discuss other `GridLayout` features as we present the GUI-building steps. To learn more about class `GridLayout`, visit:

```
http://developer.android.com/reference/android/widget/GridLayout.html
```

Id Property Values for This App's Views

Figure 3.4 shows the views' **Id** property values. For clarity, our naming convention is to use the view's class name in the view's **Id** property and Java variable name.

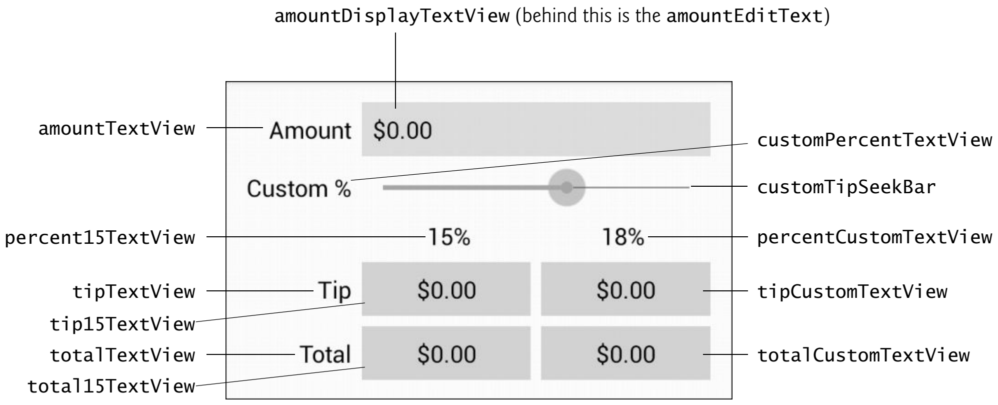


Fig. 3.4 | **Tip Calculator** GUI's components labeled with their **Id** property values.

In the right column of the first row, there are actually *two* components in the *same* grid cell—the `amountDisplayTextView` is *hiding* the `amountEditText` that receives the user input. As you'll soon see, we restrict the user's input to integer digits so that the user cannot enter invalid input. However, we want the user to see the bill amount as a *currency* value. As the user enters each digit, we divide the amount by 100.0 and display the currency-formatted result in the `amountDisplayTextView`. In the *U.S. locale*, if the user enters 3456, as each digit is entered the `amountDisplayTextView` will show the values `$0.03`, `$0.34`, `$3.45` and `$34.56`, respectively.

`LinearLayout` Id Property Values

Figure 3.5 shows the **Id**s of the three horizontal `LinearLayouts` in the `GridLayout`'s right column.

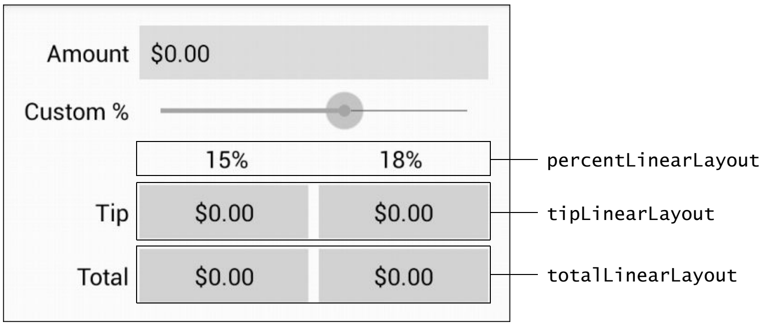


Fig. 3.5 | **Tip Calculator** GUI's LinearLayouts with their **Id** property values.

3.4.2 Creating the `TipCalculator` Project

The Android Developer Tools IDE allows only *one* project with a given name per workspace, so before you create the new project, delete the `TipCalculator` project that you test-drove in Section 3.2. To do so, right click it and select **Delete**. In the dialog that appears, ensure that **Delete project contents on disk** is *not* selected, then click **OK**. This removes the project from the workspace, but leaves the project's folder and files on disk in case you'd like to look at our original app again later.

Creating a New **Blank App** *Project*

Next, create a new **Android Application Project**. Specify the following values in the **New Android Project** dialog's first **New Android Application** step, then press **Next >**:

- **Application Name:** `Tip Calculator`
- **Project Name:** `TipCalculator`
- **Package Name:** `com.deitel.tipcalculator`
- **Minimum Required SDK:** `API18: Android 4.3`
- **Target SDK:** `API19: Android 4.4`
- **Compile With:** `API19: Android 4.4`
- **Theme:** `Holo Light with Dark Action Bar`
- **Create Activity:** `TipCalculator`
- **Build Target:** Ensure that **Android 4.3** is checked

In the **New Android Project** dialog's second **New Android Application** step, leave the default settings, then press **Next >**. In the **Configure Launcher Icon** step, click the **Browse...** button, select the `DeitelGreen.png` app icon image (provided in the `images` folder with the book's examples), click the **Open** button, then press **Next >**. In the **Create Activity** step, select **Blank Activity**, then press **Next >**. In the **Blank Activity** step, leave the default settings, then press **Finish** to create the project. Close `MainActivity.java` and `fragment_main.xml`, then open `activity_main.xml`. In the **Graphical Layout** editor, select **Nexus 4** from the screen-type drop-down list (as in Fig. 2.12). Once again, we'll use this device as the basis for our design.

3.4.3 Changing to a `GridLayout`

The default layout in `activity_main.xml` is a `FrameLayout`. Here, you'll change that to a `GridLayout`. Right click the `RelativeLayout` in the **Outline** window and select **Change Layout...**. In the **Change Layout** dialog, select `GridLayout` and click **OK**. The IDE changes the layout and sets its **Id** to `GridLayout1`. We changed this to `gridLayout` using the **Id** field in the **Properties** window. By default, the `GridLayout`'s **Orientation** property is set to horizontal, indicating that its contents will be laid out row-by-row. Ensure that the `GridLayout`'s **Padding Left** and **Padding Right** properties are set to `activity_horizontal_margin` and that the **Padding Top** and **Padding Bottom** properties are set to `activity_vertical_margin`.

Specifying Two Columns and Default Margins for the ***`GridLayout`***

Recall that the GUI in Fig. 3.3 consists of two columns. To specify this, select `gridLayout` in the **Outline** window, then change its **Column Count** property to `2` (in the **Properties** window's **GridLayout** group). By default, there are *no margins*—spaces that separate views—around a `GridLayout`'s cells. Set the `GridLayout`'s **Use Default Margins** property to `true` to indicate that the `GridLayout` should place margins around its cells. By default, the `GridLayout` uses the recommended gap between views (`8dp`), as specified at

```
http://developer.android.com/design/style/metrics-grids.html
```

3.4.4 Adding the `TextView`s, `EditText`, `SeekBar` and `LinearLayout`s

You'll now build the GUI in Fig. 3.3. You'll start with the basic layout and views in this section. In Section 3.4.5, you'll customize the views' properties to complete the design. As you add each view to the GUI, immediately set its **Id** property using the names in Figs. 3.4–3.5. You can change the selected view's **Id** via the **Properties** window or by right clicking the view (in the **Graphical Layout** editor or **Outline** window), selecting **Edit ID...** and changing the **Id** in the **Rename Resource** dialog that appears.

In the following steps, you'll use the **Outline** window to add views to the `GridLayout`. When working with layouts, it can be difficult to see the layout's *nested structure* and to place views in the correct locations by dragging them onto the **Graphical Layout** editor window. The **Outline** window makes these tasks easier because it shows the GUI's nested structure. Perform the following steps in the exact order specified—otherwise, the views will *not* appear in the correct order in each row. If this happens, you can reorder views by dragging them in the **Outline** window.

Step 1: Adding Views to the First Row

The first row consists of the `amountTextView` in the first column and the `amountEditText` behind the `amountDisplayTextView` in the second column. Each time you drop a view or layout onto the `gridLayout` in the **Outline** window, the view is placed in the layout's *next open cell*, unless you specify otherwise by setting the view's **Row** and **Column** properties. You'll do that in this step so that the `amountEditText` and `amountDisplayTextView` are placed in the same cell.

All of the `TextView`s in this app use the *medium*-sized font from the app's theme. The **Graphical Layout** editor's **Palette** provides *preconfigured* `TextView`s named **Large**, **Medium** and **Small** (in the **Form Widgets** section) to represent the theme's corresponding text sizes. In each case, the IDE configures the `TextView`'s **Text Appearance** property accordingly. Perform the following tasks to add the two `TextView`s and the `EditText`:

1. Drag a **Medium** `TextView` from the **Palette**'s **Form Widgets** section and drop it on the `gridLayout` in the **Outline** window. The IDE creates a new `TextView` named `textView1` and nests it in the `gridLayout` node. The default text `"Medium Text"` appears in the **Graphical Layout** editor. Change the `TextView`'s **Id** to `amountTextView`. You'll change its text in Step 6 (Section 3.4.5).
2. This app allows you to enter only *non-negative integers*, which the app divides by 100.0 to display the bill amount. The **Palette**'s **Text Fields** section provides many *preconfigured* `EditText`s for various forms of input (e.g., numbers, times, dates, addresses and phone numbers). When the user interacts with an `EditText`, an appropriate keyboard is displayed based on the `EditText`'s *input type*. When you hover over an `EditText` in the **Palette**, a *tooltip* indicates the input type. From the **Palette**'s **Text Fields** section, drag a **Number** `EditText` (displayed with the number **42** on it) and drop it on the `gridLayout` node in the **Outline** window. Change the

EditText's **Id** to amountEditText. The EditText is placed in the *second* column of the GridLayout's *first* row.

3. Drag another **Medium** TextView onto the gridLayout node in the **Outline** window and change the **Id** to amountDisplayTextView. The new TextView is initially placed in the *first* column of the GridLayout's *second* row. To place it in the *second* column of the GridLayout's *first* row, set this TextView's **Row** and **Column** properties (located in the **Properties** window's **Layout Parameters** section) to the values 0 and 1, respectively.

Step 2: Adding Views to the Second Row

Next, you'll add a TextView and SeekBar to the GridLayout. To do so:

1. Drag a **Medium** TextView (customPercentTextView) from the **Palette**'s **Form Widgets** section onto the gridLayout node in the **Outline** window.
2. Drag a SeekBar (customTipSeekBar) from the **Palette**'s **Form Widgets** section onto the gridLayout node in the **Outline** window.

Step 3: Adding Views to the Third Row

Next, you'll add a LinearLayout containing two TextViews to the GridLayout. To do so:

1. From the **Palette**'s **Layouts** section, drag a **Linear Layout (Horizontal)** (percentLinearLayout) onto the gridLayout node in the **Outline** window.
2. Drag a **Medium** TextView (percent15TextView) onto the percentLinearLayout node in the **Outline** window. This nests the new TextView in the LinearLayout.
3. Drag another **Medium** TextView (percentCustomTextView) onto the percentLinearLayout node in the **Outline** window.
4. The percentLinearLayout and its two nested TextViews should be placed in the second column of the GridLayout. To do so, select the percentLinearLayout in the **Outline** window, then set its **Column** property to 1.

Step 4: Adding Views to the Fourth Row

Next, you'll add a TextView and a LinearLayout containing two more TextViews to the GridLayout. To do so:

1. Drag a **Medium** TextView (tipTextView) onto the gridLayout node.
2. Drag a **Linear Layout (Horizontal)** (tipLinearLayout) onto the gridLayout node.
3. Drag two **Medium** TextViews (tip15TextView and tipCustomTextView) onto the tipLinearLayout node.

Step 5: Adding Views to the Fifth Row

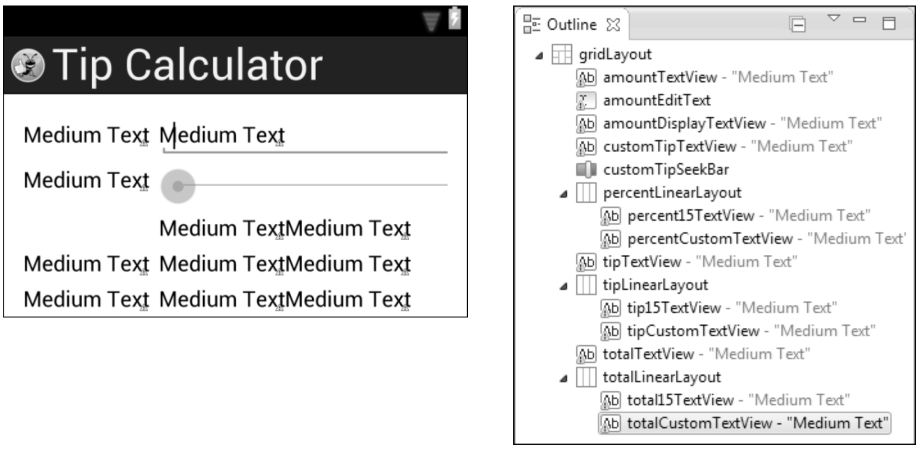
To create the last row of the GUI, repeat Step 4, using the **Id**s totalTextView, totalLinearLayout, total15TextView and totalCustomTextView.

Reviewing the Layout So Far

The GUI and **Outline** window should now appear as shown in Fig. 3.6. The warning symbols shown in the **Graphical Layout** editor and the **Outline** window will go away as you complete the GUI design in Section 3.4.5.

a) GUI design so far

b) Outline window showing **Tip Calculator** components



```
gridLayout
    amountTextView - "Medium Text"
    amountEditText
    amountDisplayTextView - "Medium Text"
    customTipTextView - "Medium Text"
    customTipSeekBar
    percentLinearLayout
        percent15TextView - "Medium Text"
        percentCustomTextView - "Medium Text"
    tipTextView - "Medium Text"
    tipLinearLayout
        tip15TextView - "Medium Text"
        tipCustomTextView - "Medium Text"
    totalTextView - "Medium Text"
    totalLinearLayout
        total15TextView - "Medium Text"
        totalCustomTextView - "Medium Text"
```

Fig. 3.6 | The GUI and the IDE's **Outline** window after adding all the views to the `GridLayout`.

3.4.5 Customizing the Views to Complete the Design

You'll now complete the app's design by customizing the views' properties and creating several string and dimension resources. As you learned in Section 2.5, literal string values should be placed in the `strings.xml` resource file. Similarly, literal numeric values that specify view dimensions (e.g., widths, heights and spacing) should be placed in the `dimens.xml` resource file.

Step 6: Specifying Literal Text

Specify the literal text for the `amountTextView`, `customPercentTextView`, `percent15TextView`, `percentCustomTextView`, `tipTextView` and `totalTextView`:

1. Select the `amountTextView` in the **Outline** window.
2. In the **Properties** window, click the ellipsis button next to the **Text** property.
3. In the **Resource Chooser Dialog**, click **New String...**.
4. In the **Create New Android String** dialog, specify `Amount` in the **String** field and `amount` in the **New R.string** field, then click **OK**.
5. In the **Resource Chooser** dialog, click **OK** to set the `amountTextView`'s **Text** property to the string resource identified as `amount`.

Repeat the preceding tasks for the other `TextView`s using the values shown in Fig. 3.7.

View	String	New R.string
`customPercentTextView`	`Custom %`	`custom_tip_percentage`
`percent15TextView`	`15%`	`fifteen_percent`

Fig. 3.7 | String resource values and resource IDs. (Part 1 of 2.)

View	String	New R.string
`percentCustomTextView`	`18%`	`eighteen_percent`
`tipTextView`	`Tip`	`tip`
`totalTextView`	`Total`	`total`

Fig. 3.7 | String resource values and resource IDs. (Part 2 of 2.)

Step 7: Right Aligning the **`TextViews`** *in the Left Column*

In Fig. 3.3, each of the left column's `TextViews` is right aligned. For the `amountTextView`, `customPercentTextView`, `tipTextView` and `totalTextView`, set the layout **Gravity** property to `right`—located in the **Layout Parameters** section in the **Properties** window.

Step 8: Configuring the **`amountTextView`***'s* **Label For** *Property*

Generally, each `EditText` should have a descriptive `TextView` that helps the user understand the `EditText`'s purpose (also helpful for accessibility)—otherwise, *Android Lint* issues a warning. To fix this, you set the `TextView`'s **Label For** property to the **Id** of the associated `EditText`. Select the `amountTextView` and set its **Label For** property (in the **Properties** window's **View** section) to

```
@+id/amountEditText
```

The + is required because the `TextView` is defined *before* the `EditText` in the GUI, so the `EditText` does not yet exist when Android converts the layout's XML into the GUI.

Step 9: Configuring the **`amountEditText`**

In the final app, the `amountEditText` is *hidden* behind the `amountDisplayTextView` and is configured to allow only *digits* to be entered by the user. Select the `amountEditText` and set the following properties:

1. In the **Properties** window's **Layout Parameters** section, set the **Width** and **Height** to `wrap_content`. This indicates that the `EditText` should be just large enough to fit its content, including any padding.
2. Remove the layout **Gravity** value `fill_horizontal`, leaving the property's value blank. We'll discuss `fill_horizontal` in the next step.
3. Remove the **Ems** property's value, which indicates the `EditText`'s width, measured in uppercase M characters of the view's font. In our `GridLayout`, this causes the second column to be too narrow, so we removed this default setting.
4. In the **Properties** window's **TextView** section, set **Digits** to `0123456789`—this allows *only* digits to be entered, even though the numeric keypad contains minus (`-`), comma (`,`), period (`.`) and space buttons. By default, the **Digits** property is *not* displayed in the **Properties** window, because it's considered to be an advanced property. To display it, click the **Show Advanced Properties** (⇶) toggle button at the top of the **Properties** window.
5. We restricted the bill amount to a maximum of *six* digits—so the largest supported bill amount is 9999.99. In the **Properties** window's **TextView** section, set the **Max Length** property to `6`.

Step 10: Configuring the `amountDisplayTextView`

To complete the formatting of the `amountDisplayTextView`, select it and set the following properties:

1. In the **Properties** window's **Layout Parameters** section, set the **Width** and **Height** to `wrap_content` to indicate that the `TextView` should be large enough to fit its content.
2. Remove the **Text** property's value—we'll programmatically display text here.
3. In the **Properties** window's **Layout Parameters** section, set the layout **Gravity** to `fill_horizontal`. This indicates that the `TextView` should occupy all remaining horizontal space in this `GridLayout` row.
4. In the **View** section, set the **Background** to `@android:color/holo_blue_bright`. This is one of several *predefined colors* (each starts with `@android:color`) in Android's *Holo* theme. As you start typing the **Background** property's value, a drop-down list of the theme's available colors is displayed. You can also use any *custom color* created from a combination of red, green and blue components called **RGB values**—each is an integer in the range 0–255 that defines the amount of red, green and blue in the color, respectively. Custom colors are defined in *hexadecimal (base 16) format*, so the RGB components are values in the range 00–FF. Android also supports *alpha* (*transparency*) values in the range 0 (*completely transparent*) to 255 (*completely opaque*). To use alpha, you specify the color in the format `#AARRGGBB`, where the first two hexadecimal digits represent the alpha value. If both digits of each color component are the same, you can use the abbreviated formats `#RGB` or `#ARGB`. For example, `#9AC` is treated as `#99AACC` and `#F9AC` is treated as `#FF99AACC`.
5. Finally, you'll add some padding around the `TextView`. To do so, you'll create a new *dimension resource* named `textview_padding`, which you'll use several times in the GUI. A view's **Padding** property specifies space on all sides of the views's content. In the **Properties** window's **View** section, click the **Padding** property's ellipsis button. Click **New Dimension...** to create a new *dimension resource*. Specify `textview_padding` for the **Name** and `8dp` for the **Value** and click **OK**, then select your new *dimension resource* and click **OK**.

Step 11: Configuring the `customPercentTextView`

Notice that the `customPercentTextView` is aligned with the top of the `customTipSeekBar`'s thumb. This looks better if it's *vertically centered*. To do this, in the **Properties** window's **Layout Parameters** section, modify the **Gravity** value from `right` to

```
    right|center_vertical
```

The *vertical bar* (`|`) character is used to separate *multiple* **Gravity** values—in this case indicating that the `TextView` should be *right aligned* and *centered vertically* within the grid cell. Also set the `customPercentTextView`'s **Width** and **Height** properties to `wrap_content`.

Step 12: Configuring the `customTipSeekBar`

By default, a `SeekBar`'s range is `0` to `100` and its current value is indicated by its **Progress** property. This app allows custom tip percentages from 0 to 30 and specifies a default of 18. Set the `SeekBar`'s **Max** property to `30` and the **Progress** property to `18`. Also, set the **Width** and **Height** to `wrap_content`.

Step 13: Configuring the **`percent15TextView`** *and* **`percentCustomTextView`**

Recall that `GridLayout` does *not* allow you to specify how a view should be sized relative to other views in a given row. This is why we placed the `percent15TextView` and `percentCustomTextView` in a `LinearLayout`, which *does* allow *proportional sizing*. A view's layout **Weight** (in certain layouts, such as `LinearLayout`) specifies the view's relative importance with respect to other views in the layout. By default, all views have a **Weight** of `0`.

In this layout, we set **Weight** to `1` for `percent15TextView` and `percentCustomTextView`—this indicates that they have equal importance, so they should be sized equally. By default, when we added the `percentLinearLayout` to the `GridLayout`, its layout **Gravity** property was set to `fill_horizontal`, so the layout occupies the remaining space in the third row. When the `LinearLayout` is stretched to fill the rest of the row, the `TextViews` each occupy *half* of the `LinearLayout`'s width.

We also wanted each `TextView` to center its text. To do this, in the **Properties** window's **TextView** section, set the **Gravity** property to `center`. This specifies the `TextView`'s text alignment, whereas the *layout* **Gravity** property specifies how a view aligns with respect to the layout.

Step 14: Configuring the **`tip15TextView`**, **`tipCustomTextView`**, **`total15TextView`** *and* **`totalCustomTextView`**

To finalize these four `TextViews`, perform the following tasks on each:

1. Select the `TextView`.
2. Delete its **Text** value—we'll set this programmatically.
3. Set the **Background** to `@android:color/holo_orange_light`.
4. Set the layout **Gravity** to `center`.
5. Set the layout **Weight** to `1`.
6. Set the layout **Width** to `0dp`—this allows the layout to use the **Weight** to determine the view's width.
7. Set the `TextView` **Gravity** to `center`.
8. Set the `TextView` **Padding** to `@dimen/textview_padding` (the *dimension resource* you created in a previous step).

Notice that there's *no horizontal space* between the `TextViews` in the `tipLinearLayout` and `totalLinearLayout`. To fix this, you'll specify an `8dp` right margin for the `tip15TextView` and `total15TextView`. In the **Properties** window's **Layout Parameters** section, expand the **Margin** section, then set the **Right** margin to `8dp` by creating a new *dimension resource* named `textview_margin`. Next, use this resource to set the `total15TextView`'s **Right** margin.

Step 15: Vertically Centering the **`tipTextView`** *and* **`totalTextView`**

To vertically center the `tipTextView` and `totalTextView` with the other views in their respective rows, modify their layout **Gravity** properties from `right` to

```
    right|center_vertical
```

When you do this for the `totalTextView`, the `GridLayout` centers this component vertically in the *remaining space from the fifth row to the bottom of the screen*. To fix this problem, drag a **Space** view (in the **Palette**'s **Layout** section) onto the `gridLayout` node in the **Outline** window. This creates a sixth row that occupies the rest of the screen. As its name implies, a **Space** view occupies space in a GUI. The GUI should now appear as in Fig. 3.8.

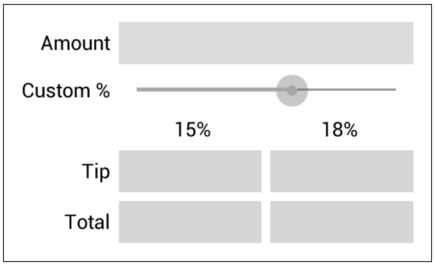


Fig. 3.8 | Final GUI design.

3.5 Adding Functionality to the App

Class `MainActivity` (Figs. 3.9–3.16) implements the **Tip Calculator** app's functionality. It calculates the 15% and custom percentage tips and total bill amounts, and displays them in locale-specific currency format. To view the file, open `src/com.deitel/tipcalculator` and double clck `MainActivity.java`. You'll need to enter most of the code in Figs. 3.9–3.16.

The `package` *and* `import` *Statements*

Figure 3.9 shows the `package` statement and `import` statements in `MainActivity.java`. The `package` statement in line 3 was inserted when you created the project. When you open a Java file in the IDE, the `import` statements are collapsed—one is displayed with a ⊕ to its left. You can click the ⊕ to see the complete list of `import` statements.

```
 1  // MainActivity.java
 2  // Calculates bills using 15% and custom percentage tips.
 3  package com.deitel.tipcalculator;
 4
 5  import java.text.NumberFormat; // for currency formatting
 6
 7  import android.app.Activity; // base class for activities
 8  import android.os.Bundle; // for saving state information
 9  import android.text.Editable; // for EditText event handling
10  import android.text.TextWatcher; // EditText listener
11  import android.widget.EditText; // for bill amount input
12  import android.widget.SeekBar; // for changing custom tip percentage
13  import android.widget.SeekBar.OnSeekBarChangeListener; // SeekBar listener
14  import android.widget.TextView; // for displaying text
15
```

Fig. 3.9 | `MainActivity`'s `package` and `import` statements.

Lines 5–14 `import` the classes and interfaces the app uses:

- Class `NumberFormat` of package `java.text` (line 5) provides numeric formatting capabilities, such as *locale-specific* currency and percentage formats.

- Class `Activity` of package `android.app` (line 7) provides the basic *lifecycle methods* of an app—we'll discuss these shortly.
- Class `Bundle` of package `android.os` (line 8) represents an app's *state information*. Android gives an app the opportunity to *save its state* before another app appears on the screen. This might occur, for example, when the user *launches another app* or *receives a phone call*. The app that's currently on the screen at a given time is in the *foreground* (the user can interact with it, and the app consumes the CPU) and all other apps are in the *background* (the user cannot interact with them, and they're typically not consuming the CPU). When another app comes into the foreground, the app that was previously in the foreground is given the opportunity to *save its state* as it's sent to the background.
- Interface `Editable` of package `android.text` (line 9) allows you to modify the content and markup of text in a GUI.
- You implement interface `TextWatcher` of package `android.text` (line 10) to respond to events when the user changes the text in an `EditText`.
- Package `android.widget` (lines 11–14) contains the *widgets* (i.e., views) and layouts that are used in Android GUIs. This app uses `EditText` (line 11), `SeekBar` (line 12) and `TextView` (line 14) widgets.
- You implement interface `SeekBar.OnSeekBarChangeListener` of package `android.widget` (line 13) to respond to the user moving the `SeekBar`'s *thumb*.

As you write code with various classes and interfaces, you can use the IDE's **Source > Organize Imports** command to let the IDE insert the `import` statements for you. For cases in which the same class or interface name appears in more than one package, the IDE will let you select the appropriate `import` statement.

Tip Calculator App `Activity` and the Activity Lifecycle

Class `MainActivity` (Figs. 3.10–3.16) is the **Tip Calculator** app's `Activity` subclass. When you created the `TipCalculator` project, the IDE generated this class as a subclass of `Activity` and provided an override of class `Activity`'s inherited `onCreate` method (Fig. 3.11). Every `Activity` subclass *must* override this method. The default code for class `MainActivity` also included an `onCreateOptionsMenu` method, which we removed because it's not used in this app. We'll discuss `onCreate` shortly.

```
16  // MainActivity class for the Tip Calculator app
17  public class MainActivity extends Activity
18  {
```

Fig. 3.10 | Class `MainActivity` is a subclass of `Activity`.

Class Variables and Instance Variables

Lines 20–32 of Fig. 3.11 declare class `MainActivity`'s variables. The `NumberFormat` objects (lines 20–23) are used to format currency values and percentages, respectively. `NumberFormat static` method `getCurrencyInstance` returns a `NumberFormat` object that formats values as currency using the device's *default locale*. Similarly, `static` method `getPercentInstance` formats values as percentages using the device's *default locale*.

```
19    // currency and percent formatters
20    private static final NumberFormat currencyFormat =
21       NumberFormat.getCurrencyInstance();
22    private static final NumberFormat percentFormat =
23       NumberFormat.getPercentInstance();
24
25    private double billAmount = 0.0; // bill amount entered by the user
26    private double customPercent = 0.18; // initial custom tip percentage
27    private TextView amountDisplayTextView; // shows formatted bill amount
28    private TextView percentCustomTextView; // shows custom tip percentage
29    private TextView tip15TextView; // shows 15% tip
30    private TextView total15TextView; // shows total with 15% tip
31    private TextView tipCustomTextView; // shows custom tip amount
32    private TextView totalCustomTextView; // shows total with custom tip
33
```

Fig. 3.11 | `MainActivity` class's instance variables.

The bill amount entered by the user into `amountEditText` will be read and stored as a `double` in `billAmount` (line 25). The custom tip percentage (an integer in the range 0–30) that the user sets by moving the `Seekbar` *thumb* will be multiplied by 0.01 to create a `double` for use in calculations, then stored in `customPercent` (line 26). For example, if you select `25` with the `SeekBar`, `customPercent` will store `0.25`, so the app will multiply the bill amount by `0.25` to calculate the 25% tip.

Line 27 declares the `TextView` that displays the currency-formatted bill amount. Line 28 declares the `TextView` that displays the custom tip percentage based on the `SeekBar` *thumb's* position (see the **18%** in Fig. 3.1(a)). The variables in line 29–32 will refer to the `TextViews` in which the app displays the calculated tips and totals.

Overriding Method **`onCreate`** *of Class* **`Activity`**

The `onCreate` method (Fig. 3.12)—which is *auto-generated* with lines 38–39 when you create the app's project—is called by the system when an `Activity` is *started*. Method `onCreate` typically initializes the `Activity`'s instance variables and views. This method should be as simple as possible so that the app *loads quickly*. In fact, if the app takes longer than *five seconds* to load, the operating system will display an **ANR (Application Not Responding) dialog**—giving the user the option to *forcibly terminate the app*. You'll learn how to prevent this problem in Chapter 8.

```
34    // called when the activity is first created
35    @Override
36    protected void onCreate(Bundle savedInstanceState)
37    {
38       super.onCreate(savedInstanceState); // call superclass's version
39       setContentView(R.layout.activity_main); // inflate the GUI
40
```

Fig. 3.12 | Overriding `Activity` method `onCreate`. (Part 1 of 2.)

```
41       // get references to the TextViews
42       // that MainActivity interacts with programmatically
43       amountDisplayTextView =
44          (TextView) findViewById(R.id.amountDisplayTextView);
45       percentCustomTextView =
46          (TextView) findViewById(R.id.percentCustomTextView);
47       tip15TextView = (TextView) findViewById(R.id.tip15TextView);
48       total15TextView = (TextView) findViewById(R.id.total15TextView);
49       tipCustomTextView = (TextView) findViewById(R.id.tipCustomTextView);
50       totalCustomTextView =
51          (TextView) findViewById(R.id.totalCustomTextView);
52
53       // update GUI based on billAmount and customPercent
54       amountDisplayTextView.setText(
55          currencyFormat.format(billAmount));
56       updateStandard(); // update the 15% tip TextViews
57       updateCustom(); // update the custom tip TextViews
58
59       // set amountEditText's TextWatcher
60       EditText amountEditText =
61          (EditText) findViewById(R.id.amountEditText);
62       amountEditText.addTextChangedListener(amountEditTextWatcher);
63
64       // set customTipSeekBar's OnSeekBarChangeListener
65       SeekBar customTipSeekBar =
66          (SeekBar) findViewById(R.id.customTipSeekBar);
67       customTipSeekBar.setOnSeekBarChangeListener(customSeekBarListener);
68    } // end method onCreate
69
```

Fig. 3.12 | Overriding `Activity` method `onCreate`. (Part 2 of 2.)

`onCreate`'s `Bundle` Parameter

During the app's execution, the user could change the device's configuration by *rotating the device* or *sliding out a hard keyboard*. For a good experience, the app should continue operating smoothly through such configuration changes. When the system calls `onCreate`, it passes a **`Bundle`** argument containing the `Activity`'s saved state, if any. Typically, you save state in `Activity` methods `onPause` or `onSaveInstanceState` (demonstrated in later apps). Line 38 calls the superclass's `onCreate` method, which is *required* when overriding `onCreate`.

Generated `R` Class Contains Resource IDs

As you build your app's GUI and add *resources* (such as `strings` in the `strings.xml` file or views in the `activity_main.xml` file) to your app, the IDE generates a class named **`R`** that contains *nested classes* representing each type of resource in your project's `res` folder. You can find this class in your project's **gen folder**, which contains generated source-code files. The nested classes are declared `static`, so that you can access them in your code with `R.`*ClassName*. Within class `R`'s nested classes, the IDE creates `static final int` constants that enable you to refer to your app's resources programmatically from your code (as we'll discuss momentarily). Some of the nested classes in class `R` include:

- class **`drawable`**—contains constants for any `drawable` items, such as *images*, that you put in the various `drawable` folders in your app's `res` folder

- class **`id`**—contains constants for the *views* in your *XML layout files*
- class **`layout`**—contains constants that represent each *layout file* in your project (such as, `activity_main.xml`)
- class **`string`**—contains constants for each `String` in the `strings.xml` file.

Inflating the GUI

The call to **`setContentView`** (line 39) receives the constant **`R.layout.activity_main`** to indicate which XML file represents `MainActivity`'s GUI—in this case, the constant represents the `main.xml` file. Method `setContentView` uses this constant to load the corresponding XML document, which is then parsed and converted into the app's GUI. This process is known as **inflating** the GUI.

Getting References to the Widgets

Once the layout is *inflated*, you can *get references to the individual widgets* so that you can interact with them programmatically. To do so, you use class `Activity`'s `findViewById` method. This method takes an `int` constant representing a specific view's **Id** and returns a reference to the view. The name of each view's **`R.id`** constant is determined by the component's **Id** property that you specified when designing the GUI. For example, `amountEditText`'s constant is `R.id.amountEditText`.

Lines 43–51 obtain references to the `TextView`s that are changed by the app. Lines 43–44 obtain a reference to the `amountDisplayTextView` that's updated when the user enters the bill amount. Lines 45–46 obtain a reference to the `percentCustomTextView` that's updated when the user changes the custom tip percentage. Lines 47–51 obtain references to the `TextView`s where the calculated tips and totals are displayed.

Displaying Initial Values in the `TextViews`

Lines 54–55 set `amountDisplayTextView`'s text to the initial `billAmount` (`0.00`) in a *locale-specific* currency format by calling the `currencyFormat` object's **`format` method**. Next, lines 56–57 call methods `updateStandard` (Fig. 3.13) and `updateCustom` (Fig. 3.14) to display initial values in the tip and total `TextView`s.

Registering the Event Listeners

Lines 60–61 get a reference to the `amountEditText`, and line 62 calls its `addTextChangedListener` method to register the `TextChangedListener` that will respond to *events* generated when the *user changes the text* in the `EditText`. We define this listener (Fig. 3.16) as an *anonymous-inner-class object* that's assigned to the instance variable `amountEditTextWatcher`.

Lines 65–66 get a reference to the `customTipSeekBar` and line 67 calls its `setOnSeekBarChangeListener` method to register the `OnSeekBarChangeListener` that will respond to *events* generated when the user moves the `customTipSeekBar`'s *thumb* to change the custom tip percentage. We define this listener (Fig. 3.15) as an *anonymous-inner-class object* that's assigned to the instance variable `customSeekBarListener`.

Method `updateStandard` *of Class* `MainActivity`

Method `updateStandard` (Fig. 3.13) updates the 15% tip and total `TextView`s each time the user *changes* the bill amount. The method uses the `billAmount` value to calculate the tip amount and the total of the bill amount and tip. Lines 78–79 display the amounts in currency format.

```
70      // updates 15% tip TextViews
71      private void updateStandard()
72      {
73         // calculate 15% tip and total
74         double fifteenPercentTip = billAmount * 0.15;
75         double fifteenPercentTotal = billAmount + fifteenPercentTip;
76
77         // display 15% tip and total formatted as currency
78         tip15TextView.setText(currencyFormat.format(fifteenPercentTip));
79         total15TextView.setText(currencyFormat.format(fifteenPercentTotal));
80      } // end method updateStandard
81
```

Fig. 3.13 | Method `updateStandard` calculates and displays the 15% tip and total.

Method `updateCustom` *of Class* `MainActivity`

Method `updateCustom` (Fig. 3.14) updates the custom tip and total `TextViews` based on the tip percentage the user selected with the `customTipSeekBar`. Line 86 sets the `percent-CustomTextView`'s text to the `customPercent` value formatted as a percentage. Lines 89–90 calculate the `customTip` and `customTotal`. Then, lines 93–94 display the amounts in currency format.

```
82      // updates the custom tip and total TextViews
83      private void updateCustom()
84      {
85         // show customPercent in percentCustomTextView formatted as %
86         percentCustomTextView.setText(percentFormat.format(customPercent));
87
88         // calculate the custom tip and total
89         double customTip = billAmount * customPercent;
90         double customTotal = billAmount + customTip;
91
92         // display custom tip and total formatted as currency
93         tipCustomTextView.setText(currencyFormat.format(customTip));
94         totalCustomTextView.setText(currencyFormat.format(customTotal));
95      } // end method updateCustom
96
```

Fig. 3.14 | Method `updateCustom` calculates and displays the custom tip and total.

Anonymous Inner Class That Implements Interface `OnSeekBarChangeListener`

Lines 98–120 of Fig. 3.15 create the *anonymous-inner-class* object named `customSeekBar-Listener` that responds to `customTipSeekBar`'s *events*. If you're not familiar with *anonymous inner classes*, visit the following page:

```
http://bit.ly/AnonymousInnerClasses
```

Line 67 (Fig. 3.12) registered `customSeekBarListener` as `customTipSeekBar`'s `OnSeek-BarChangeListener` *event handler*. For clarity, we define all but the simplest event-handling objects in this manner so that we do not clutter the `onCreate` method with this code.

```
 97     // called when the user changes the position of SeekBar
 98     private OnSeekBarChangeListener customSeekBarListener =
 99        new OnSeekBarChangeListener()
100        {
101           // update customPercent, then call updateCustom
102           @Override
103           public void onProgressChanged(SeekBar seekBar, int progress,
104              boolean fromUser)
105           {
106              // sets customPercent to position of the SeekBar's thumb
107              customPercent = progress / 100.0;
108              updateCustom(); // update the custom tip TextViews
109           } // end method onProgressChanged
110
111           @Override
112           public void onStartTrackingTouch(SeekBar seekBar)
113           {
114           } // end method onStartTrackingTouch
115
116           @Override
117           public void onStopTrackingTouch(SeekBar seekBar)
118           {
119           } // end method onStopTrackingTouch
120        }; // end OnSeekBarChangeListener
121
```

Fig. 3.15 | Anonymous inner class that implements interface `OnSeekBarChangeListener` to respond to the events of the `customSeekBar`.

Overriding Method `onProgressChanged` *of Interface* `OnSeekBarChangeListener`

Lines 102–119 implement interface `OnSeekBarChangeListener`'s methods. Method `onProgressChanged` is called whenever the `SeekBar`'s *thumb* position *changes*. Line 107 calculates `customPercent` using the method's `progress` parameter—an `int` representing the `SeekBar`'s *thumb* position. We divide this by 100.0 to get the custom percentage. Line 108 calls method `updateCustom` to recalculate and display the custom tip and total.

Overriding Methods `onStartTrackingTouch` *and* `onStopTrackingTouch` *of Interface* `OnSeekBarChangeListener`

Java requires that you override *every* method in an *interface* that you *implement*. This app does *not* need to know when the user *starts* moving the slider's thumb (`onStartTrackingTouch`) or *stops* moving it (`onStopTrackingTouch`), so we simply provide an *empty* body for each (lines 111–119) to *fulfill* the *interface contract*.

Anonymous Inner Class That Implements Interface `TextWatcher`

Lines 123–156 of Fig. 3.16 create the *anonymous-inner-class* object `amountEditTextWatcher` that responds to `amountEditText`'s *events*. Line 62 registered this object to *listen* for `amountEditText`'s events that occur when the text changes.

Overriding Method `onTextChanged` *of Interface* `TextWatcher`

The `onTextChanged` method (lines 126–144) is called whenever the text in the `amountEditText` is *modified*. The method receives four parameters. In this example, we use only

```
122     // event-handling object that responds to amountEditText's events
123     private TextWatcher amountEditTextWatcher = new TextWatcher()
124     {
125        // called when the user enters a number
126        @Override
127        public void onTextChanged(CharSequence s, int start,
128           int before, int count)
129        {
130           // convert amountEditText's text to a double
131           try
132           {
133              billAmount = Double.parseDouble(s.toString()) / 100.0;
134           } // end try
135           catch (NumberFormatException e)
136           {
137              billAmount = 0.0; // default if an exception occurs
138           } // end catch
139
140           // display currency formatted bill amount
141           amountDisplayTextView.setText(currencyFormat.format(billAmount));
142           updateStandard(); // update the 15% tip TextViews
143           updateCustom(); // update the custom tip TextViews
144        } // end method onTextChanged
145
146        @Override
147        public void afterTextChanged(Editable s)
148        {
149        } // end method afterTextChanged
150
151        @Override
152        public void beforeTextChanged(CharSequence s, int start, int count,
153           int after)
154        {
155        } // end method beforeTextChanged
156     }; // end amountEditTextWatcher
157  } // end class MainActivity
```

Fig. 3.16 | Anonymous inner class that implements interface `TextWatcher` to respond to the events of the `amountEditText`.

`CharSequence s`, which contains a copy of `amountEditText`'s text. The other parameters indicate that the `count` characters starting at `start` *replaced* previous text of length `before`.

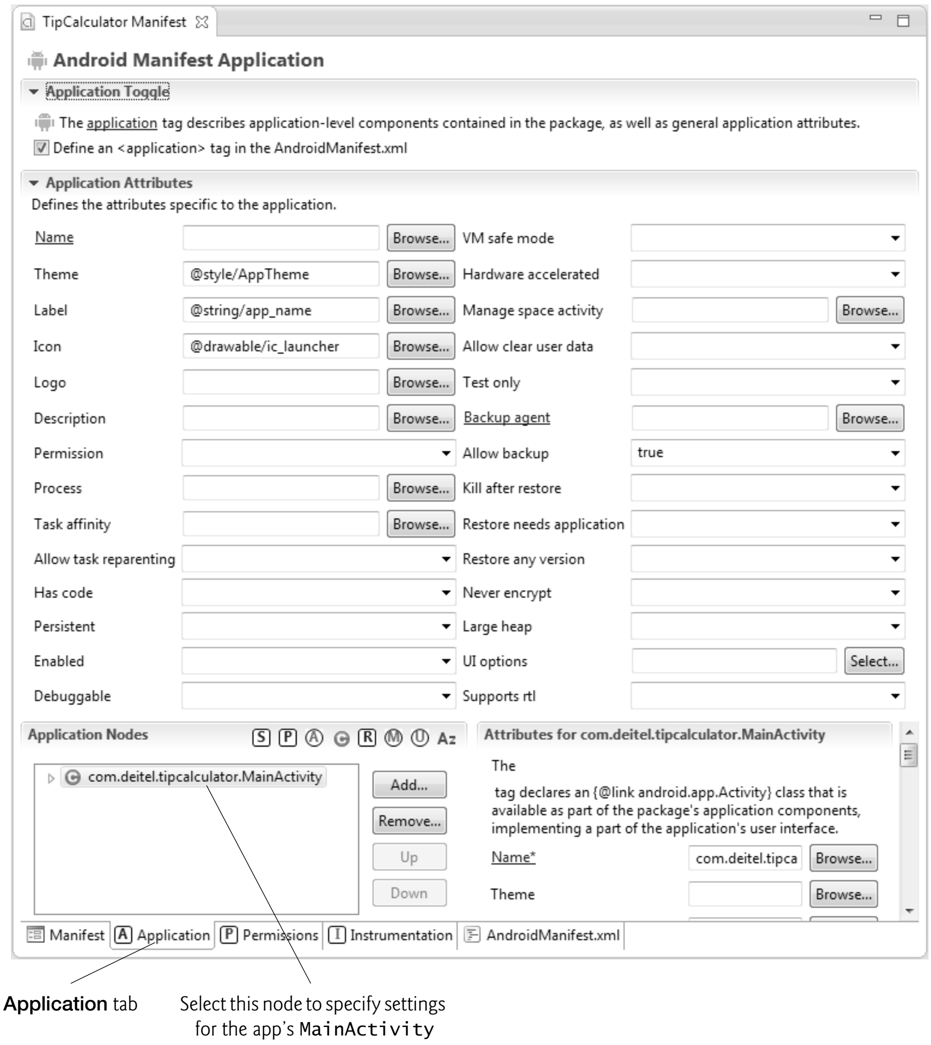
Line 133 converts the user input from `amountEditText` to a `double`. We allow users to enter only whole numbers in pennies, so we divide the converted value by 100.0 to get the actual bill amount—e.g., if the user enters 2495, the bill amount is 24.95. Lines 142–143 call `updateStandard` and `updateCustom` to recalculate and display the tips and totals.

Other Methods of the `amountEditTextWatcher` `TextWatcher`

This app does *not* need to know what changes are about to be made to the text (`beforeTextChanged`) or that the text has already been changed (`afterTextChanged`), so we simply override each of these `TextWatcher` interface methods with an *empty* body (lines 146–155) to *fulfill the interface contract*.

3.6 AndroidManifest.xml

In this section, you'll modify the `AndroidManifest.xml` file to specify that this app's `Activity` supports only a device's *portrait* orientation and that the *soft keypad* should *always* remain on the screen. You'll use the IDE's **Android Manifest** editor to specify these settings. To open the **Android Manifest** editor, double click the app's `AndroidManifest.xml` file in the **Package Explorer**. At the bottom of the editor, click the **Application** tab (Fig. 3.17), then select the `MainActivity` node in the **Application Nodes** section at the bottom of the window. This displays settings for the `MainActivity` in the **Attributes for com.deitel.tipcalculator.MainActivity** section.


TipCalculator Manifest

Android Manifest Application

Application Toggle

The application tag describes application-level components contained in the package, as well as general application attributes.

☑ Define an <application> tag in the AndroidManifest.xml

Application Attributes

Defines the attributes specific to the application.

Name		VM safe mode	
Theme	@style/AppTheme	Hardware accelerated	
Label	@string/app_name	Manage space activity	
Icon	@drawable/ic_launcher	Allow clear user data	
Logo		Test only	
Description		Backup agent	
Permission		Allow backup	true
Process		Kill after restore	
Task affinity		Restore needs application	
Allow task reparenting		Restore any version	
Has code		Never encrypt	
Persistent		Large heap	
Enabled		UI options	
Debuggable		Supports rtl	

Application Nodes

com.deitel.tipcalculator.MainActivity

Attributes for com.deitel.tipcalculator.MainActivity

The

tag declares an {@link android.app.Activity} class that is available as part of the package's application components, implementing a part of the application's user interface.

Name*	com.deitel.tipca
Theme	

Fig. 3.17 | **Android Manifest** editor's **Application** tab.

*Configuring **MainActivity** for Portrait Orientation*

In general, most apps should support *both* portrait and landscape orientations. In *portrait* orientation, the device's height is greater than its width. In *landscape orientation*, the device's width is greater than its height. In the **Tip Calculator** app, rotating the device to landscape orientation on a typical phone would cause the numeric keypad to obscure most of the **Tip Calculator**'s GUI. For this reason, you'll configure `MainActivity` to support *only* portrait orientation. In the **Android Manifest** editor's **Attributes for com.deitel.tipcalculator.MainActivity** section, scroll down to the **Screen orientation** option and select `portrait`.

*Forcing the Soft Keypad to Always Display for **MainActivity***

In the **Tip Calculator** app, the soft keypad should be displayed immediately when the app executes and should remain on the screen at all times. In the **Android Manifest** editor's **Attributes for com.deitel.tipcalculator.MainActivity** section, scroll down to the **Window soft input mode** option and select `stateAlwaysVisible`. Note that this will *not* display the soft keyboard if a hard keyboard is present.

3.7 Wrap-Up

In this chapter, you created your first *interactive* Android app—the **Tip Calculator**. We overviewed the app's capabilities, then you test-drove it to calculate standard and custom tips based on the bill amount entered. You followed detailed step-by-step instructions to build the app's GUI using the Android Developer Tools IDE's **Graphical Layout** editor, **Outline** window and **Properties** window. We also walked through the code of the `Activity` subclass `MainActivity`, which defined the app's functionality.

In the app's GUI, you used a `GridLayout` to arrange the views into rows and columns. You displayed text in `TextViews` and received input from an `EditText` and a `SeekBar`.

The `MainActivity` class required many Java object-oriented programming capabilities, including classes, objects, methods, interfaces, anonymous inner classes and inheritance. We explained the notion of inflating the GUI from its XML file into its screen representation. You learned about Android's `Activity` class and part of the `Activity` lifecycle. In particular, you overrode the `onCreate` method to initialize the app when it's launched. In the `onCreate` method, you used `Activity` method `findViewById` to get references to each of the views that the app interacts with programmatically. You defined an anonymous inner class that implements the `TextWatcher` interface so the app can calculate new tips and totals as the user changes the text in the `EditText`. You also defined an anonymous inner class that implements the `OnSeekBarChangeListener` interface so the app can calculate a new custom tip and total as the user changes the custom tip percentage by moving the `SeekBar`'s thumb.

Finally, you opened the `AndroidManifest.xml` file in the IDE's **Android Manifest** editor to specify that the `MainActivity` supports only portrait orientation and that the `MainActivity` should always display the keypad.

Using the IDE's **Graphical Layout** editor, **Outline** window, **Properties** window and **Android Manifest** editor enabled you to build this app without manipulating the XML in the project's resource files and `AndroidManifest.xml` file.

In the next chapter, we introduce collections while building the **Twitter® Searches** app. Many mobile apps display lists of items. You'll do this by using a `ListActivity` containing a `ListView` that's bound to an `ArrayList<String>`. You'll also store app data as user preferences and learn how to launch the device's web browser to display a web page.

Self-Review Exercises

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

a) A __________ can arrange views either *horizontally* or *vertically* and size its views proportionally.
b) Use a(n) __________ to arrange GUI components into cells in a rectangular grid.
c) When working with more complex layouts like `GridLayouts`, it's difficult to see the nested structure of the layout and to place components in the correct nested locations using the Visual Layout Editor. The __________ window makes these tasks easier because it shows the nested structure of the GUI. So, in a `GridLayout`, you can select the appropriate row and add a GUI component to it.
d) Class __________ of package `android.app` provides the basic *lifecycle methods* of an app.
e) You implement interface __________ of package `android.text` to respond to events when the user interacts with an `EditText` component.
f) An __________ activity is always visible and "has the focus."
g) The method __________ is called by the system when an `Activity` is starting—that is, when its GUI is about to be displayed so that the user can interact with the `Activity`.
h) As you build your app's GUI and add resources (such as `strings` in the `strings.xml` file or GUI components in the `activity_main.xml` file) to your app, the IDE generates a class named __________ that contains nested `static` classes representing each type of resource in your project's `res` folder.
i) Class __________ (nested in class `R`)—contains constants for any `drawable` items, such as images, that you put in the various `drawable` folders in your app's `res` folder.
j) Class __________ (nested in class `R`)—contains constants for each `String` in the `strings.xml` file.
k) Once the layout is inflated, you can get references to the individual widgets using `Activity`'s __________ method. This method takes an `int` constant for a specific view (that is, a GUI component) and returns a reference to it.

3.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Method `onCreate` typically initializes the `Activity`'s instance variables and GUI components. This method should be as simple as possible so that the app loads quickly. In fact, if the app takes longer than five seconds to load, the operating system will display an ANR (Application Not Responding) dialog—giving the user the option to forcibly terminate the app.
b) To center a **TextView**'s text, in the **Properties** window of **TextView**, set the layout **Gravity** property to center.
c) As with all Java programs, Android apps have a main method.
d) Each `Activity` lifecycle method you override need not call the superclass's version of the same method.
e) A stopped `activity` is visible on the screen and is likely to be killed by the system when its memory is needed.

Answers to Self-Review Exercises

3.1 a) `LinearLayout`. b) `GridLayout`. c) **Outline**. d) Activity. e) `TextWatcher`. f) active g) `onCreate`. h) `R`. i) `R.drawable`. j) `R.string`. k) `findViewById`.

3.2 a) True. b) False. Layout **Gravity** is used for setting children views' alignment within a parent `LinearLayout`. c) False. Android apps don't have a main method. d) False. Each `Activity` lifecycle method you override must call the superclass's version; otherwise, an exception will occur. e) False. A stopped activity is *not* visible on the screen and is likely to be killed by the system when its memory is needed.

Exercises

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

a) XML Layout files should be placed in the __________ folder of the project.
b) The widgets and layouts that are used in Android GUIs are found in Package __________.
c) The `TextView`'s __________ property specifies how a view aligns with respect to the layout.
d) Interface __________ of package `android.text` allows you to change the content and markup of text in a GUI.
e) You implement interface __________ of package `android.widget` to respond to the user moving the `SeekBar`'s thumb.
f) Android apps have four types of components—activities, services, content providers and __________.
g) A __________ activity is visible on the screen but does *not* have the focus—such as when an alert dialog is displayed.
h) Class __________ (nested in class `R`)—contains constants for the GUI components in your XML layout files.
i) The device's height is greater than its width in __________ orientation. In __________ orientation, the device's width is greater than its height.

3.4 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `NumberFormat` class is used to create locale-specific currency and percentage strings.
b) A GUI component can span multiple columns in a `GridLayout`.
c) Every `Activity` subclass must override the `onCreate` method.
d) A paused activity is visible on the screen and has the focus.
e) A `GridLayout` can specify within a given row that the horizontal space should be allocated proportionally between multiple views.
f) You override the `onStart` method to initialize the app when it's launched.

3.5 *(Enhanced* **Tip Calculator** *App)* Make the following enhancements to the **Tip Calculator** app:

a) Add an option to calculate the tip based on either the price before tax or after tax.
b) Allow the user to enter the number of people in the party. Calculate and display the amount owed by each person if the bill were to be split evenly among the party members.

3.6 *(**EMI Calculator** App)* Create an EMI calculator app that allows the user to enter a loan amount, an interest rate and the number of years. Based on these values, the app should calculate the amount to be paid every month and display the monthly payment for the number of years entered. Allow the user to select a custom loan duration (in years) by using a `SeekBar` and display the monthly payment for that custom loan duration.

3.7 *(**Housing Loan Interest Calculator** App)* A bank offers housing loans that can be repaid in 7, 14 or 21 years. Write an app that allows the user to enter the amount of the loan and the annual interest rate. Based on these values, the app should display a `Seekbar` to select the loan lengths in years, monthly payment and amount paid towards interest every year.

3.8 *(**Car Payment Calculator** App)* Typically, banks offer car loans for periods ranging from two to five years (24 to 60 months). Borrowers repay the loans in monthly installments. The amount of each monthly payment is based on the length of the loan, the amount borrowed and the interest rate. Create an app that allows the customer to enter the price of a car, the down-payment amount and the loan's annual interest rate. The app should display the loan's duration in months and the monthly payments for two-, three-, four- and five-year loans. The variety of options allows the user to easily compare repayment plans and choose the most appropriate.

3.9 ***(Miles-Per-Gallon Calculator App)*** Drivers often want to know the miles per gallon their cars get so they can estimate gasoline costs. Develop an app that allows the user to input the number of miles driven and the number of gallons used and calculates and displays the corresponding miles per gallon.

3.10 ***(Body Mass Index Calculator App)*** The formulas for calculating the BMI are

$$BMI = \frac{weightInPounds \times 703}{heightInInches \times heightInInches}$$

or

$$BMI = \frac{weightInKilograms}{heightInMeters \times heightInMeters}$$

Create a BMI calculator app that allows users to enter their weight and height and whether they are entering these values in English or Metric units, then calculates and displays the user's body mass index. The app should also display the following information from the Department of Health and Human Services/National Institutes of Health so the user can evaluate his/her BMI:

```
BMI VALUES
Underweight: less than 18.5
Normal:      between 18.5 and 24.9
Overweight:  between 25 and 29.9
Obese:       30 or greater
```

3.11 ***(Target-Heart-Rate Calculator App)*** While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA), the formula for calculating your *maximum heart rate* in beats per minute is *220 minus your age in years* (`http://bit.ly/AHATargetHeartRates`). Your *target heart rate* is a range that is 50–85% of your maximum heart rate. [*Note:* These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.] Write an app that inputs the person's age, then calculates and displays the person's maximum heart rate and target-heart-rate range.

Twitter® Searches App

4

Objectives

In this chapter you'll:

- Support both portrait and landscape device orientations.
- Extend `ListActivity` to create an `Activity` that displays a list of items in a `ListView`.
- Enable users to interact with an app via an `ImageButton`.
- Manipulate collections of data.
- Use `SharedPreferences` to store key–value pairs of data associated with an app.
- Use a `SharedPreferences.Editor` to modify key–value pairs of data associated with an app.
- Use an `ArrayAdapter` to specify a `ListView`'s data.
- Use an `AlertDialog.Builder` object to create dialogs that display options as `Button`s or in a `ListView`.
- Use an implicit `Intent` to open a website in a browser.
- Use an implicit `Intent` to display an intent chooser containing a list of apps that can share text.
- Programmatically hide the soft keyboard.

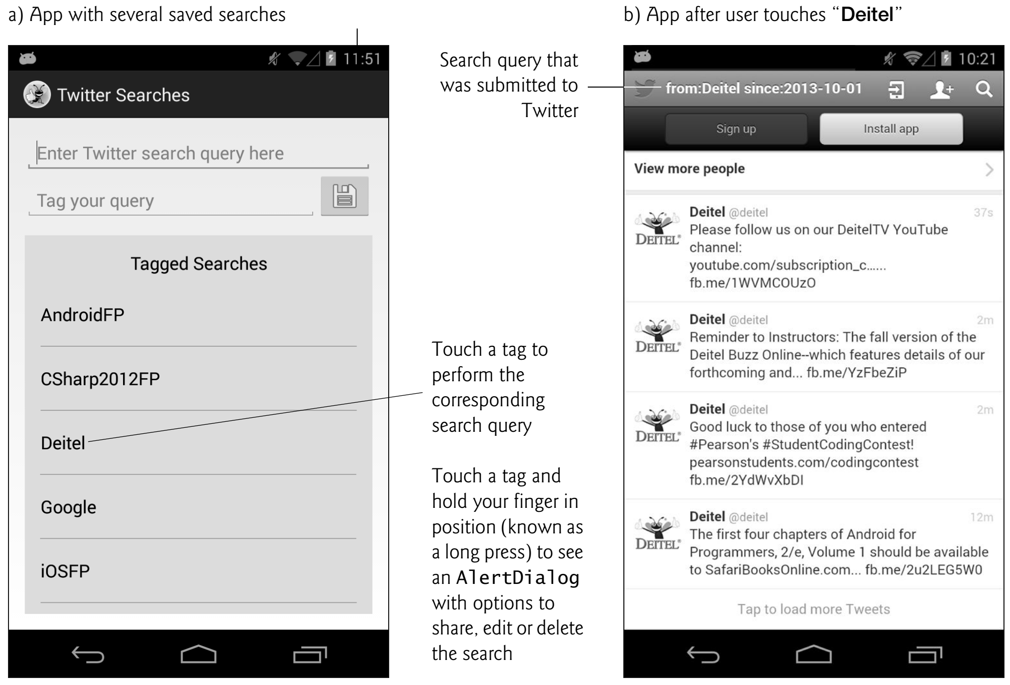
Outline

4.1 Introduction

Twitter's search mechanism makes it easy to follow trending topics being discussed by more than 500 million Twitter users. Searches can be fine tuned using Twitter's *search operators* (overviewed in Section 4.2), often resulting in lengthy search strings that are time consuming and cumbersome to enter on a mobile device. The **Twitter Searches** app (Fig. 4.1) allows you to save your favorite search queries with short tag names that are easy to remember (Fig. 4.1(a)). You can then touch a tag name to quickly and easily follow tweets on a given topic (Fig. 4.1(b)). As you'll see, the app also allows you to *share, edit* and *delete* saved searches.

The app supports both portrait and landscape device orientations. In some apps, you'll do this by providing separate layouts for each orientation. In this app, we support both orientations by designing the GUI so that it can dynamically adjust GUI component sizes based on the current orientation.

First, you'll test-drive the app. Then we'll overview the technologies used to build it. Next, you'll design the app's GUI. Finally, we'll present the app's complete source code and walk through the code, discussing the app's new features in more detail.


a) App with several saved searches
b) App after user touches "**Deitel**"
Search query that was submitted to Twitter
Touch a tag to perform the corresponding search query
Touch a tag and hold your finger in position (known as a long press) to see an `AlertDialog` with options to share, edit or delete the search

Twitter Searches

Enter Twitter search query here

Tag your query

Tagged Searches

AndroidFP

CSharp2012FP

Deitel

Google

iOSFP

from:Deitel since:2013-10-01

Sign up | Install app

View more people

Deitel @deitel 37s
Please follow us on our DeitelTV YouTube channel: youtube.com/subscription_c...... fb.me/1WVMCOUzO

Deitel @deitel 2m
Reminder to Instructors: The fall version of the Deitel Buzz Online--which features details of our forthcoming and... fb.me/YzFbeZiP

Deitel @deitel 2m
Good luck to those of you who entered #Pearson's #StudentCodingContest! pearsonstudents.com/codingcontest fb.me/2YdWvXbDI

Deitel @deitel 12m
The first four chapters of Android for Programmers, 2/e, Volume 1 should be available to SafariBooksOnline.com... fb.me/2u2LEG5W0

Tap to load more Tweets

Fig. 4.1 | **Twitter Searches** app.

4.2 Test-Driving the App

In this section, you'll test-drive the **Twitter Searches** app. Open the Android Developer Tools IDE and import the **Twitter Searches** app project. As you did in Chapter 3, launch the Nexus 4 AVD—or connect your Android device to the computer—so that you can test the app. The screen captures we show in this chapter were taken on a Nexus 4 phone.

4.2.1 Importing the App and Running It

Perform the following steps to import the app into the IDE:

1. *Opening the* **Import** *dialog.* Select **File > Import....**
2. *Importing the* **Twitter Searches** *app's project.* Expand the **General** node and select **Existing Projects into Workspace**. Click **Next >** to proceed to the **Import Projects** step. Ensure that **Select root directory** is selected, then click **Browse....** Locate the `TwitterSearches` folder in the book's examples folder, select it and click **OK**. Ensure that **Copy projects into workspace** is *not* selected. Click **Finish** to import the project so that it appears in the **Package Explorer** window.
3. *Launching the* **Twitter Searches** *app.* Right click the `TwitterSearches` project in the **Package Explorer** window, then select **Run As > Android Application** to execute **Twitter Searches** in the AVD or on your device. This builds the project and runs the app (Fig. 4.2).

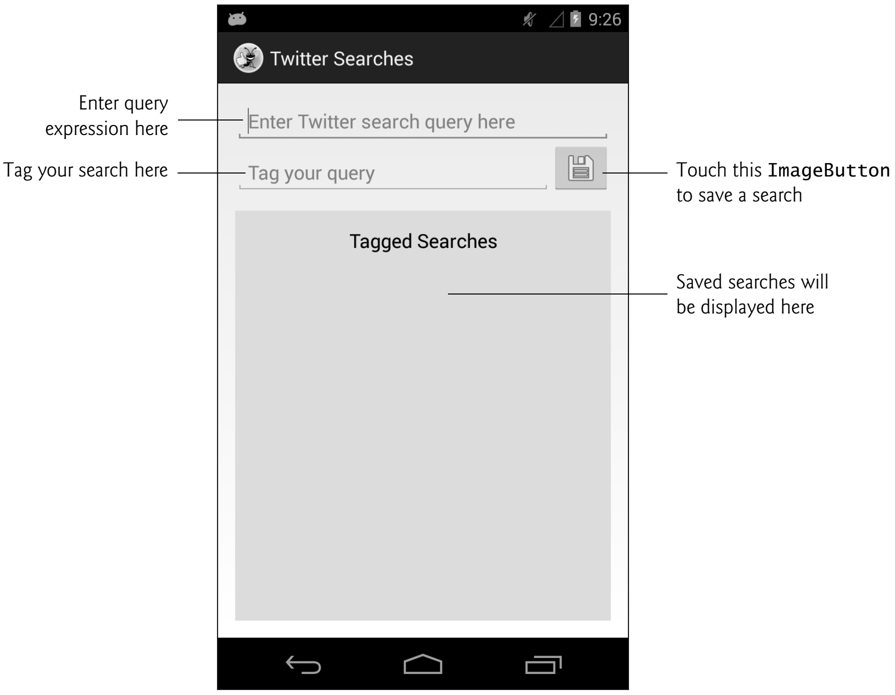

9:26
Twitter Searches
Enter Twitter search query here
Tag your query
Tagged Searches
Enter query expression here
Tag your search here
Touch this ImageButton to save a search
Saved searches will be displayed here

Fig. 4.2 | **Twitter Searches** app when it first executes.

4.2.2 Adding a Favorite Search

Touch the top `EditText`, then enter `from:deitel` as the search query—the `from:` operator locates tweets from a specified Twitter account. Figure 4.3 shows several Twitter search operators—multiple operators can be used to construct more complex queries. A complete list can be found at

```
http://bit.ly/TwitterSearchOperators
```

Example	Finds tweets containing
`deitel iOS6`	Implicit *logical and* operator—Finds tweets containing `deitel` *and* `iOS6`.
`deitel OR iOS6`	Logical `OR` operator—Finds tweets containing `deitel` *or* `iOS6` *or both*.
`"how to program"`	String in quotes("")—Finds tweets containing the exact phrase `"how to program"`.
`deitel ?`	`?` (question mark)—Finds tweets asking questions about `deitel`.
`deitel -sue`	`-` (minus sign)—Finds tweets containing `deitel` but not `sue`.
`deitel :)`	`:)` (happy face)—Finds *positive attitude* tweets containing `deitel`.
`deitel :(`	`:(` (sad face)—Finds *negative attitude* tweets containing `deitel`.

Fig. 4.3 | Some Twitter search operators. (Part 1 of 2.)

Example	Finds tweets containing
`since:2013-10-01`	Finds tweets that occurred *on or after* the specified date, which must be in the form `YYYY-MM-DD`.
`near:"New York City"`	Finds tweets that were sent near `"New York City"`.
`from:deitel`	Finds tweets from the Twitter account `@deitel`.
`to:deitel`	Finds tweets to the Twitter account `@deitel`.

Fig. 4.3 | Some Twitter search operators. (Part 2 of 2.)

In the bottom `EditText` enter `Deitel` as the tag for the search query (Fig. 4.4(a)). This will be the *short name* displayed in a list in the app's **Tagged Searches** section. Touch the *save* (💾) button to save the search—the tag "**Deitel**" appears in the list under the **Tagged Searches** heading (Fig. 4.4(b)). When you save a search, the soft keyboard is dismissed so that you can see your list of saved searches—you'll learn how to programmatically hide the soft keyboard in Section 4.5.5.

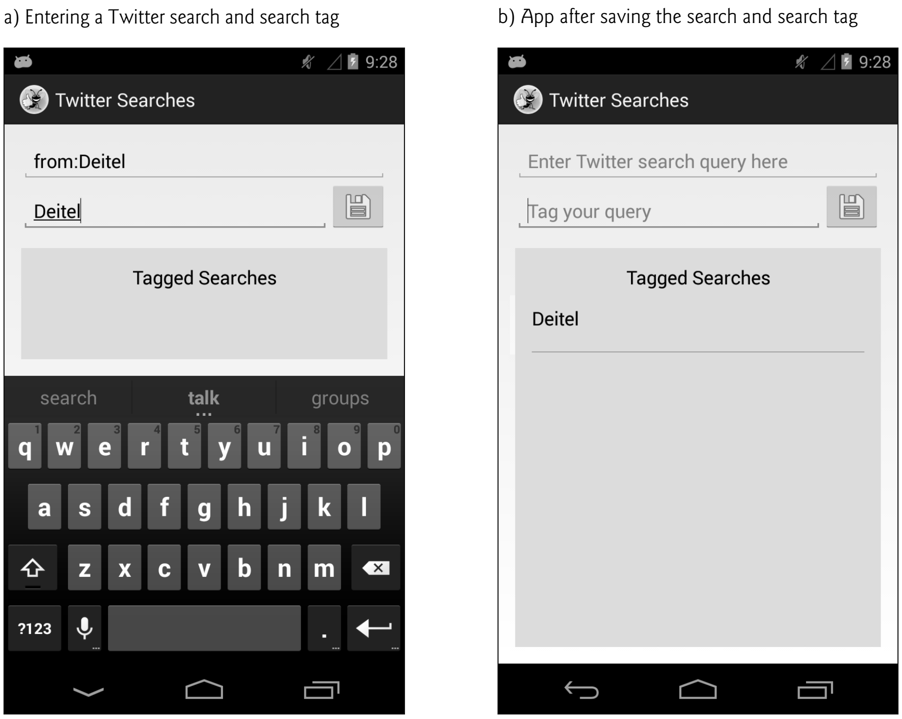
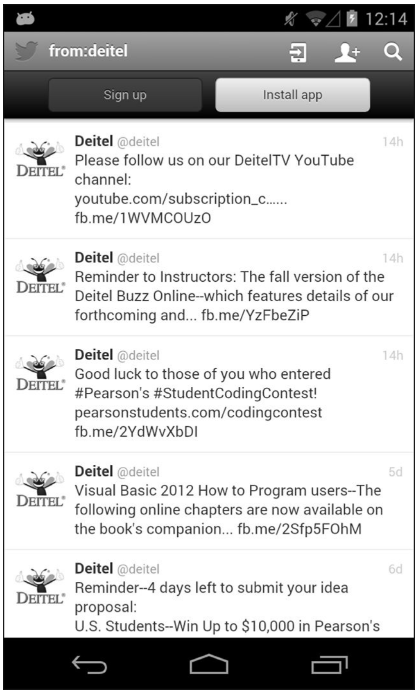


Fig. 4.4 | Entering a Twitter search.

4.2.3 Viewing Twitter Search Results

To view the search results, touch the tag "**Deitel**." This launches the device's web browser and passes a URL that represents the saved search to the Twitter website. Twitter obtains the search query from the URL, then returns the tweets that match the query (if any) as a web page. The web browser then displays this results page (Fig. 4.5). When you're done viewing the results, touch the back button ( ) to return to the **Twitter Searches** app where you can save more searches, and edit, delete and share previously saved searches.



from:deitel

Deitel @deitel 14h
Please follow us on our DeitelTV YouTube channel:
youtube.com/subscription_c......
fb.me/1WVMCOUzO

Deitel @deitel 14h
Reminder to Instructors: The fall version of the Deitel Buzz Online--which features details of our forthcoming and... fb.me/YzFbeZiP

Deitel @deitel 14h
Good luck to those of you who entered #Pearson's #StudentCodingContest!
pearsonstudents.com/codingcontest
fb.me/2YdWvXbDI

Deitel @deitel 5d
Visual Basic 2012 How to Program users--The following online chapters are now available on the book's companion... fb.me/2Sfp5FOhM

Deitel @deitel 6d
Reminder--4 days left to submit your idea proposal:
U.S. Students--Win Up to $10,000 in Pearson's

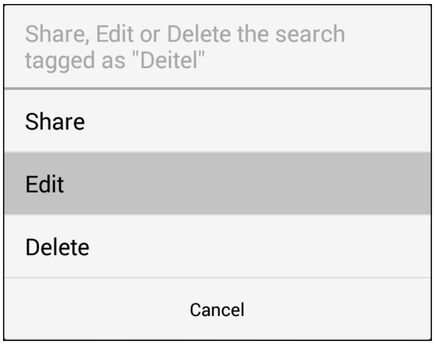
Fig. 4.5 | Viewing search results.

4.2.4 Editing a Search

You may also *share*, *edit* or *delete* a search. To see these options, *long press* the search's tag—that is, touch the tag and keep your finger on the screen. If you're using an AVD, click and hold the left mouse button on the search tag to perform a long press. When you long press "**Deitel**," the `AlertDialog` in Fig. 4.6(a) displays the **Share**, **Edit** and **Delete** options for the search tagged as "**Deitel**." If you don't wish to perform any of these tasks, touch **Cancel**.

To edit the search tagged as "**Deitel**," touch the dialog's **Edit** option. The app then loads the search's query and tag into the `EditText`s for editing. Let's restrict our search to tweets since October 1, 2013 by adding `since:2013-10-01` to the end of the query (Fig. 4.6(b)) in the top `EditText`. The `since:` operator restricts the search results to tweets that occurred *on or after* the specified date (in the form `yyyy-mm-dd`). Touch the *save* ( ) button to update the saved search, then view the updated results (Fig. 4.7) by touching **Deitel** in the **Tagged Searches** section of the app. [*Note:* Changing the tag name will create a *new* search—this is useful if you want to create a new query that's based on a previously saved query.]

a) Selecting **Edit** to edit an existing search



b) Editing the "**Deitel**" saved search

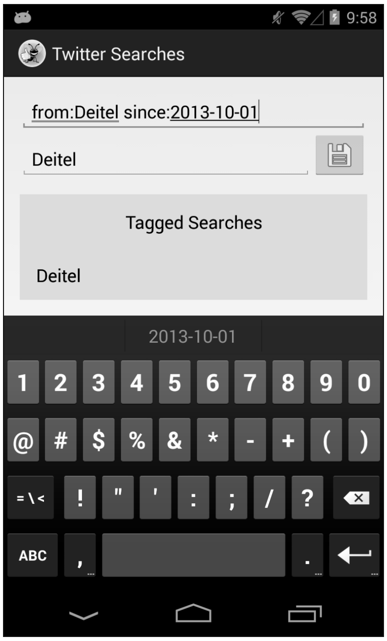
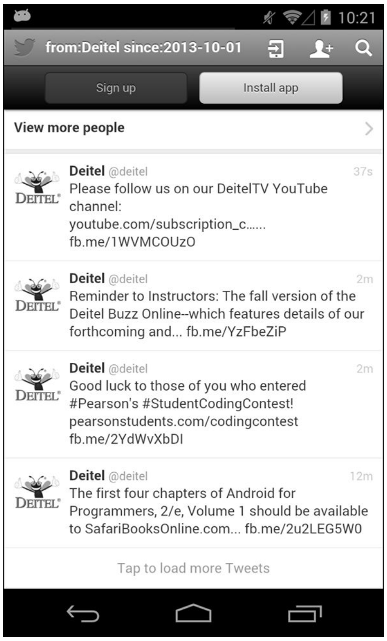


Fig. 4.6 | Editing a saved search.



Deitel @deitel 37s
Please follow us on our DeitelTV YouTube channel: youtube.com/subscription_c...... fb.me/1WVMCOUzO

Deitel @deitel 2m
Reminder to Instructors: The fall version of the Deitel Buzz Online--which features details of our forthcoming and... fb.me/YzFbeZiP

Deitel @deitel 2m
Good luck to those of you who entered #Pearson's #StudentCodingContest! pearsonstudents.com/codingcontest fb.me/2YdWvXbDI

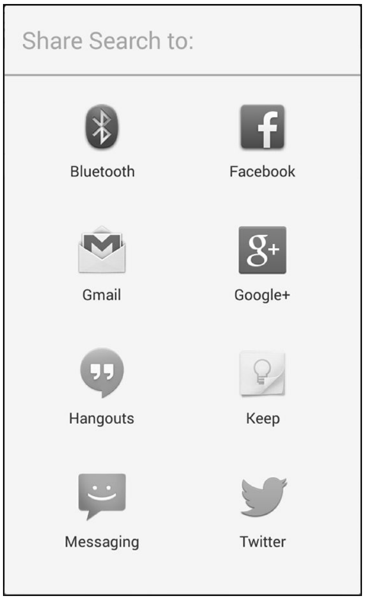
Deitel @deitel 12m
The first four chapters of Android for Programmers, 2/e, Volume 1 should be available to SafariBooksOnline.com... fb.me/2u2LEG5W0

Fig. 4.7 | Viewing the updated "**Deitel**" search results.

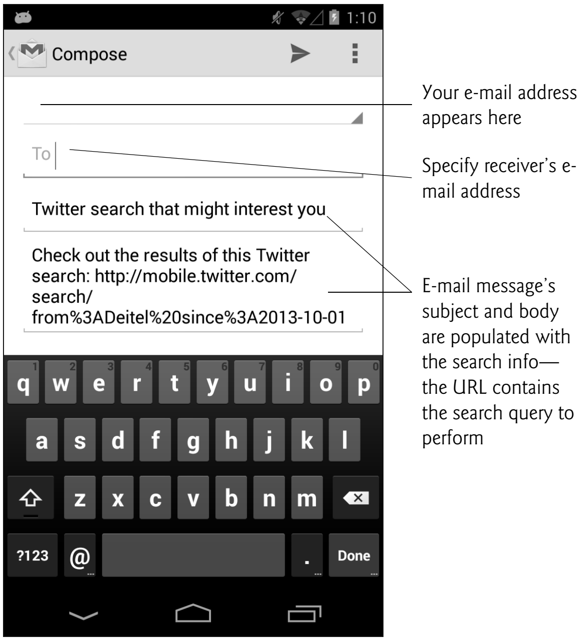
4.2.5 Sharing a Search

Android makes it easy for you to share various types of information from an app via e-mail, instant messaging (SMS), Facebook, Google+, Twitter and more. In this app, you can share a favorite search by *long pressing* the search's tag and selecting **Share** from the `AlertDialog` that appears. This displays a so-called *intent chooser* (Fig. 4.8(a)), which can vary based on the type of content you're sharing and the apps that can handle that content. In this app we're sharing text, and the intent chooser on our phone (not the AVD) shows apps capable of handling text, such as **Facebook**, **Gmail**, **Google+**, **Messaging** (instant messaging) and **Twitter**. If no apps can handle the content, the intent chooser will display a message saying so. If only one app can handle the content, that app will launch without you having to select from the intent chooser which app to use. Figure 4.8(b) shows the Gmail app's **Compose** screen with the e-mail subject and body populated. Gmail also shows your e-mail address above the **To** field (we deleted the e-mail address for privacy in the screen capture).

a) Intent chooser showing share options



b) Gmail app **Compose** screen for an e-mail containing the "**Deitel**" search


1:10
Compose
To
Your e-mail address appears here
Specify receiver's e-mail address
E-mail message's subject and body are populated with the search info—the URL contains the search query to perform
?123
Done

Twitter search that might interest you

Check out the results of this Twitter search: http://mobile.twitter.com/search/from%3ADeitel%20since%3A2013-10-01

Fig. 4.8 | Sharing a search via e-mail.

4.2.6 Deleting a Search

To delete a search, *long press* the search's tag and select **Delete** from the `AlertDialog` that appears. The app prompts you to confirm that you'd like to delete the search (Fig. 4.9)—touching **Cancel** returns you to the main screen *without* deleting the search. Touching **Delete** deletes the search.

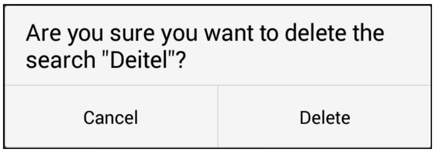

Are you sure you want to delete the search "Deitel"?

Fig. 4.9 | `AlertDialog` confirming a delete.

4.2.7 Scrolling Through Saved Searches

Figure 4.10 shows the app after we've saved 10 favorite searches—only five of which are currently visible. The app allows you to scroll through your favorite searches if there are more than can be displayed on the screen at once. The GUI component that displays the list of searches is a `ListView` (discussed in Section 4.3.1). To scroll, *drag* or *flick* your finger (or the mouse in an AVD) up or down in the list of **Tagged Searches**. Also, rotate the device to *landscape* orientation to see that the GUI dynamically adjusts.

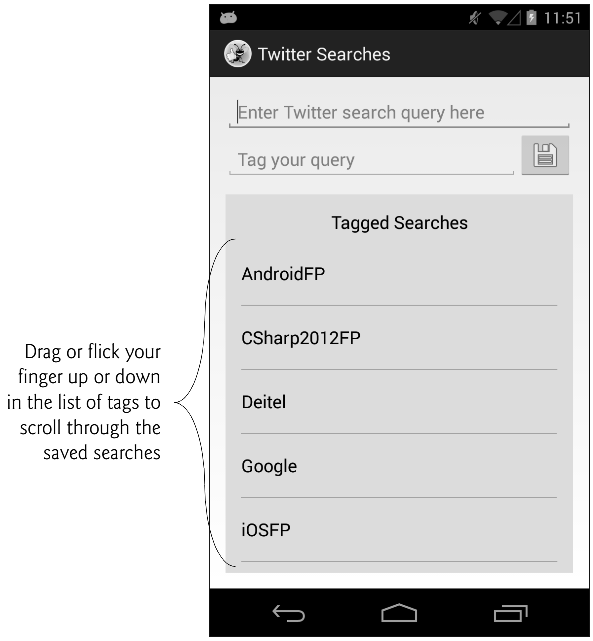

11:51
Twitter Searches
Enter Twitter search query here
Tag your query
Tagged Searches
AndroidFP
CSharp2012FP
Deitel
Google
iOSFP
Drag or flick your finger up or down in the list of tags to scroll through the saved searches

Fig. 4.10 | App with more searches than can be displayed on the screen.

4.3 Technologies Overview

This section introduces the features you'll use to build the **Twitter Searches** app.

4.3.1 `ListView`

Many mobile apps display lists of information. For example, an e-mail app displays a list of new e-mails, an address-book app displays a list of contacts, a news app displays a list of headlines, etc. In each case, the user touches an item in the list to see more information—e.g., the content of the selected e-mail, the details of the selected contact or the text of the selected news story. This app uses a **`ListView`** (package `android.widget`) to display a list of tagged searches that is *scrollable* if the complete list cannot be displayed on the screen. You can specify how to format each `ListView` item. For this app, we'll display each search's tag as a `String` in a `TextView`. In later apps, you'll completely customize the content that's displayed for each `ListView` item—displaying images, text and `Buttons`.

4.3.2 `ListActivity`

When an `Activity`'s primary task is to display a *scrollable* list of items, you can extend class **`ListActivity`** (package `android.app`), which uses a `ListView` that occupies the entire app as its default layout. `ListView` is a subclass of **`AdapterView`** (package `android.widget`)—a GUI component is bound to a data source via an **`Adapter`** object (package `android.widget`). In this app, we use an **`ArrayAdapter`** (package `android.widget`) to create an object that populates the `ListView` using data from an `ArrayList` collection object. This is known as **data binding**. Several types of `AdapterViews` can be bound to data using an `Adapter`. In Chapter 8, you'll learn how to bind database data to a `ListView`. For more details on data binding in Android and several tutorials, visit

```
http://developer.android.com/guide/topics/ui/binding.html
```

4.3.3 Customizing a `ListActivity`'s Layout

A `ListActivity`'s default GUI contains only a `ListView` that fills the screen's client area between Android's top and bottom system bars (which were explained in Fig. 2.1). If a `ListActivity`'s GUI requires only the default `ListView`, then you do *not* need to define a separate layout for your `ListActivity` subclass.

The **Twitter Searches** app's `MainActivity` displays several GUI components. For this reason you'll define a *custom* layout for `MainActivity`. When customizing a `ListActivity` subclass's GUI, the layout *must* contain a `ListView` with its **Id** attribute set to `"@android:id/list"`—the name that class `ListActivity` uses to reference its `ListView`.

4.3.4 `ImageButton`

Users often touch buttons to initiate actions in a program. To save a search's query–tag pair in this app, you touch an **`ImageButton`** (package `android.widget`). `ImageButton` is a subclass of `ImageView` which provides additional capabilities that enable an image to be used like a `Button` object (package `android.widget`) to initiate an action.

4.3.5 `SharedPreferences`

You can have one or more files containing *key–value* pairs associated with each app—each *key* enables you to quickly look up a corresponding *value*. We use this capability to manipulate a file called `searches` in which we store the pairs of tags (the *keys*) and Twitter search queries (the *values*) that the user creates. To read the key–value pairs from this file we'll use **`SharedPreferences`** objects (package **`android.content`**). To modify the file's contents, we'll use **`SharedPreferences.Editor`** objects (package `android.content`). The keys in the file must be `Strings`, and the values can be `Strings` or primitive-type values.

This app reads the saved searches in the `Activity`'s `onCreate` method—this is acceptable only because the amount of data being loaded is small. When an app is launched, Android creates a main thread called the UI thread which handles all of the GUI interactions. All GUI processing must be performed in this thread. *Extensive input/output operations, such as loading data from files and databases should not be performed on the UI thread, because such operations can affect your app's responsiveness.* We'll show how to perform I/O in separate threads in later chapters.

4.3.6 `Intents` for Launching Other Activities

Android uses a technique known as **intent messaging** to communicate information between activities within one app or activities in separate apps. Each `Activity` can specify **intent filters** indicating *actions* the `Activity` is capable of handling. Intent filters are defined in the `AndroidManifest.xml` file. In fact, in each app so far, the IDE created an intent filter for the app's only `Activity` indicating that it could respond to the predefined action named `android.intent.action.MAIN`, which specifies that the `Activity` can be used to *launch* the app to begin its execution.

An **`Intent`** is used to launch an `Activity`—it indicates an *action* to be performed and the *data* on which to perform that action. In this app, when the user touches a search tag, we create a URL that contains the Twitter search query. We load the URL into a web browser by creating a new `Intent` for viewing a URL, then passing that `Intent` to the **`startActivity` method**, which our app inherits indirectly from class `Activity`. To view a URL, `startActivity` launches the device's web browser to display the content—in this app, the results of a Twitter search.

We also use an `Intent` and the `startActivity` method to display an **intent chooser**—a GUI that shows a list of apps that can handle the specified `Intent`. We use this when sharing a saved search to allow the user to choose how to share a search.

Implicit and Explicit `Intents`

The `Intents` used in this app are examples of **implicit `Intents`**—*we will not specify a component to display the web page but instead will allow Android to launch the most appropriate `Activity` based on the type of data.* If *multiple* activities can handle the action and data passed to `startActivity`, the system will display a *dialog* in which the user can select which activity to use. If the system cannot find an activity to handle the action, then method `startActivity` throws an `ActivityNotFoundException`. In general, it's a good practice to handle this exception. We chose not to in this app, because Android devices on which this app is likely to be installed will have a browser capable of displaying a web page. In future apps, we'll also use **explicit `Intents`**, which indicate the precise `Activity` to start. For a more information on `Intents`, visit

```
http://developer.android.com/guide/components/intents-filters.html
```

4.3.7 `AlertDialog`

You can display messages, options and confirmations to app users via **`AlertDialogs`**. While a dialog is displayed, the user cannot interact with the app—this is known as a **modal dialog**. As you'll see, you specify the settings for the dialog with an **`AlertDialog.Builder`** object, then use it to create the `AlertDialog`.

AlertDialogs can display buttons, checkboxes, radio buttons and lists of items that the user can touch to respond to the dialog's message. A standard AlertDialog may have up to three buttons that represent:

- A *negative* action—Cancels the dialog's specified action, often labeled with **Cancel** or **No**. This is the leftmost button when there are multiple buttons in the dialog.
- A *positive* action—Accepts the dialog's specified action, often labeled with **OK** or **Yes**. This is the rightmost button when there are multiple buttons in the dialog.
- A *neutral* action—This button indicates that the user does not want to cancel or accept the action specified by the dialog. For example, an app that asks the user to register to gain access to additional features might provide a **Remind Me Later** neutral button.

We use AlertDialogs in this app for several purposes:

- To display a message to the user if either or both of the query and tag EditTexts are empty. This dialog will contain only a positive button.
- To display the **Share**, **Edit** and **Delete** options for a search. This dialog will contain a list of options and a negative button.
- To have the user confirm before deleting a search—in case the user accidentally touched the **Delete** option for a search.

You can learn more about Android dialogs at:

```
http://developer.android.com/guide/topics/ui/dialogs.html
```

4.3.8 AndroidManifest.xml

As you learned in Chapter 3, the AndroidManifest.xml file is created for you when you create an app. For this app, we'll show you how to add a setting to the manifest that prevents the soft keyboard from displaying when the app first loads. For the complete details of AndroidManifest.xml, visit:

```
http://developer.android.com/guide/topics/manifest/
   manifest-intro.html
```

We'll cover various aspects of the AndroidManifest.xml file throughout the book.

4.4 Building the App's GUI

In this section, we'll build the GUI for the **Twitter Searches** app. We'll also create a second XML layout that the ListView will dynamically inflate and use to display each item.

4.4.1 Creating the Project

Recall that the Android Developer Tools IDE allows only *one* project with a given name per workspace, so before you create the new project, delete the TwitterSearches project that you test-drove in Section 4.2. To do so, right click it and select **Delete**. In the dialog that appears, ensure that **Delete project contents on disk** is *not* selected, then click **OK**. This removes the project from the workspace, but leaves the project's folder and files on disk in case you'd like to look at the original app again later.

Creating a New **Blank App** *Project*

Next, create a new **Android Application Project.** Specify the following values in the **New Android Project** dialog's first **New Android Application** step, then press **Next >**:

- **Application name:** `Twitter Searches`
- **Project name:** `TwitterSearches`
- **Package name:** `com.deitel.twittersearches`
- **Minimum Required SDK:** `API18: Android 4.3`
- **Target SDK:** `API19: Android 4.4`
- **Compile With:** `API19: Android 4.4`
- **Theme:** `Holo Light with Dark Action Bar`

In the **New Android Project** dialog's second **New Android Application** step, leave the default settings, and press **Next >**. In the **Configure Launcher Icon** step, click the **Browse...** button, and select an app icon image (provided in the `images` folder with the book's examples), press **Open** then **Next >**. In the **Create Activity** step, select **Blank Activity**, then press **Next >**. In the **Blank Activity** step, leave the default settings and click **Finish** to create the project. Open `activity_main.xml` in the **Graphical Layout** editor and select **Nexus 4** from the screen-type drop-down list (as in Fig. 2.12). Once again, we'll use this device as the basis for our design.

4.4.2 `activity_main.xml` Overview

As in Chapter 3, this app's `activity_main.xml` layout uses a `GridLayout` (Fig. 4.11). In this app, the `GridLayout` contains three rows and one column. Figure 4.12 shows the names of the app's GUI components.

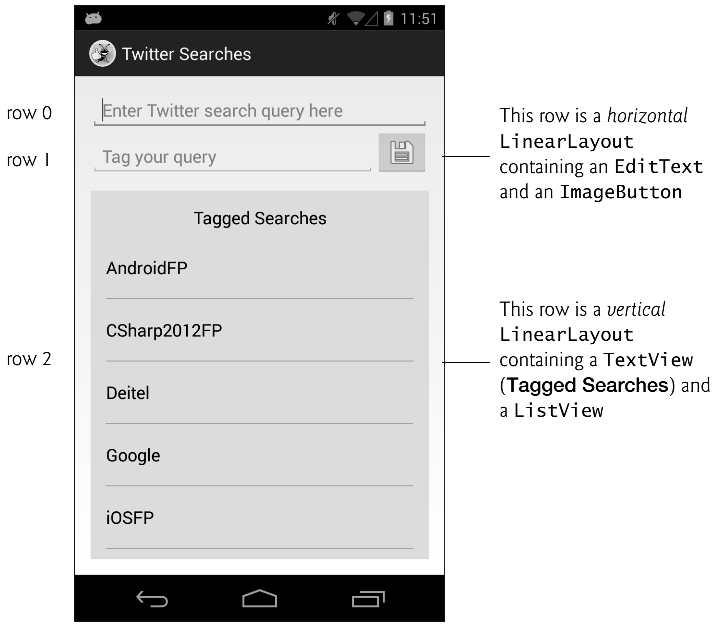

11:51
Twitter Searches
row 0
Enter Twitter search query here
row 1
Tag your query
Tagged Searches
AndroidFP
CSharp2012FP
row 2
Deitel
Google
iOSFP
This row is a *horizontal* `LinearLayout` containing an `EditText` and an `ImageButton`
This row is a *vertical* `LinearLayout` containing a `TextView` (**Tagged Searches**) and a `ListView`

Fig. 4.11 | Rows and columns in the **Twitter Searches** app's `GridLayout`.

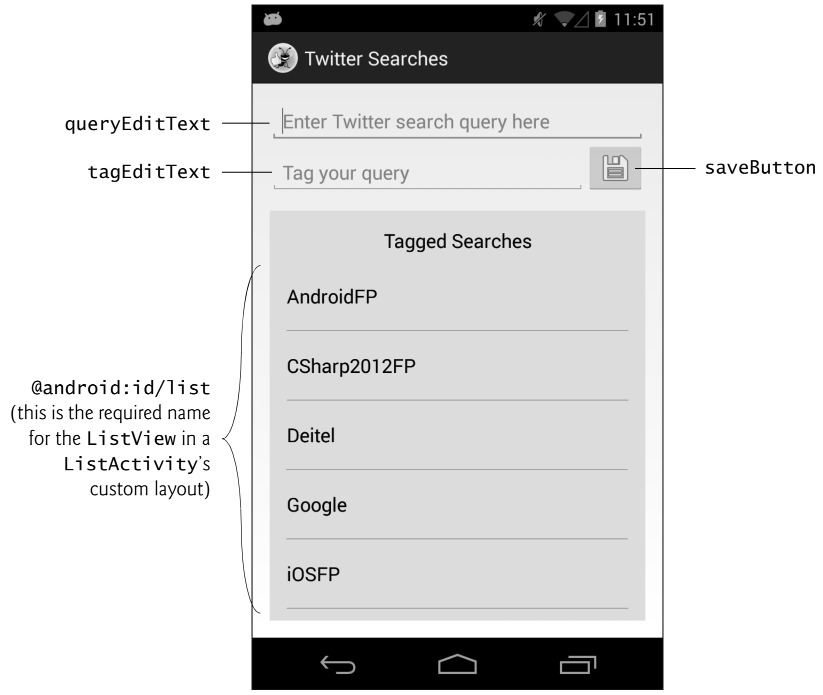


Fig. 4.12 | **Twitter Searches** GUI's components labeled with their **Id** property values.

4.4.3 Adding the `GridLayout` and Components

Using the techniques you learned in Chapter 3, you'll build the GUI in Figs. 4.11–4.12. All of the steps in the following subsections assume that you're working with the layout in the IDE's **Graphical Layout** editor. As a reminder, it's often easiest to select a particular GUI component in the **Outline** window.

You'll start with the basic layout and controls, then customize the controls' properties to complete the design. Use the **Outline** window to add components to the proper rows of the `GridLayout`. As you add GUI components, set their **Id**s as shown in Fig. 4.12—there are several components in this layout that do not require **Id**s, as they're never referenced from the app's Java code. Also, remember to define all your literal strings values in the `strings.xml` file (located in the app's `res/values` folder).

Step 1: Changing to a `GridLayout`

Follow the steps in Section 3.4.3 to switch from a `FrameLayout` to a `GridLayout`.

Step 2: Configuring the `GridLayout`

In the **Outline** window, select the `GridLayout` and set the following properties—for each property that's nested in a node within the **Properties** window, we specify the node's name in parentheses following the property name:

- **Id**: `@+id/gridLayout`

- **Column Count** (**GridLayout** node): 1—Each GUI component nested directly in the `GridLayout` will be added as a new row.

The `GridLayout` fills the entire client area of the screen because the layout's **Width** and **Height** properties (in the **Layout Parameters** section of the **Properties** window) are each set to `match_parent` by the IDE.

By default, the IDE sets the **Padding Left** and **Padding Right** properties to `@dimen/activity_horizontal_margin`—a predefined dimension resource in the `dimens.xml` file of the project's `res/values` folder. This resource's value is `16dp`, so there will be a `16dp` space to the left and right of the `GridLayout`. The IDE created this resource when you created the app's project. Similarly, the IDE sets the **Padding Top** and **Padding Bottom** properties to `@dimen/activity_vertical_margin`—another predefined dimension resource with the value `16dp`. So there will be a `16dp` space above and below the `GridLayout`.

Look-and-Feel Observation 4.1

According to the Android design guidelines, `16dp` *is the recommended space between the edges of a device's touchable screen area and the app's content; however, many apps (such as games) use the full screen.*

Step 3: Creating the `GridLayout`'s First Row

This row contains only an `EditText`. Drag a **Plain Text** component from the **Palette**'s **Text Fields** section onto the `GridLayout` in the **Outline** window, then set its **Id** property to `@+id/queryEditText`. In the **Properties** window's **TextView** node, delete the **Ems** property's value, which is not used in this app. Then use the **Properties** window to set the following properties:

- **Width** (**Layout Parameters** node): `wrap_content`
- **Height** (**Layout Parameters** node): `wrap_content`
- **Gravity** (**Layout Parameters** node): `fill_horizontal`—This ensures that when the user rotates the device, the `queryEditText` will fill all available horizontal space. We use similar **Gravity** settings for other GUI components to support both portrait and landscape orientations for this app's GUI.
- **Hint**: `@string/queryPrompt`—Create a `String` resource as you did in prior apps and give it the value `"Enter Twitter search query here"`. This attribute displays in an *empty* `EditText` a hint that helps the user understand the `EditText`'s purpose. This text is also spoken by Android TalkBack for users with visual impairments, so providing hints in your `EditTexts` makes your app more accessible.

Look-and-Feel Observation 4.2

The Android design guidelines indicate that text displayed in your GUI should be brief, simple and friendly with the important words first. For details on the recommended writing style, see `http://developer.android.com/design/style/writing.html`.

- **IME Options** (**TextView** node): `actionNext`—This value indicates that `queryEditText`'s keyboard will contain a **Next** button that the user can touch to move the input focus to the next input component (i.e., the `tagEditText` in this app). This makes it easier for the user to fill in multiple input components in a form. When the next component is another `EditText`, the appropriate keyboard is displayed without the user having to touch the `EditText` to give it the focus.

Step 4: Creating the `GridLayout`'s Second Row

This row is a horizontal `LinearLayout` containing an `EditText` and an `ImageButton`. Perform the following tasks to build the row's GUI:

1. Drag a **LinearLayout (Horizontal)** component from the **Palette**'s **Layouts** section onto the `GridLayout` in the **Outline** window.
2. Drag a **Plain Text** component from the **Palette**'s **Text Fields** section onto the `LinearLayout`, then set the **Id** property to `@+id/tagEditText`.
3. Drag an **ImageButton** component from the **Palette**'s **Images & Media** section onto the `LinearLayout`. This displays the **Resource Chooser** dialog (Fig. 4.13), so that you can choose the button's image. By default, the dialog's **Project Resources** radio button is selected so that you can choose images from the project's resources (such images would be stored in your project's various `res/drawable` folders). In this app, we used the standard Android save icon (shown at the right side of Fig. 4.13). To do so, click the **System Resources** radio button, select `ic_menu_save` and click **OK**. Next, set the **Id** property to `@+id/saveButton`.

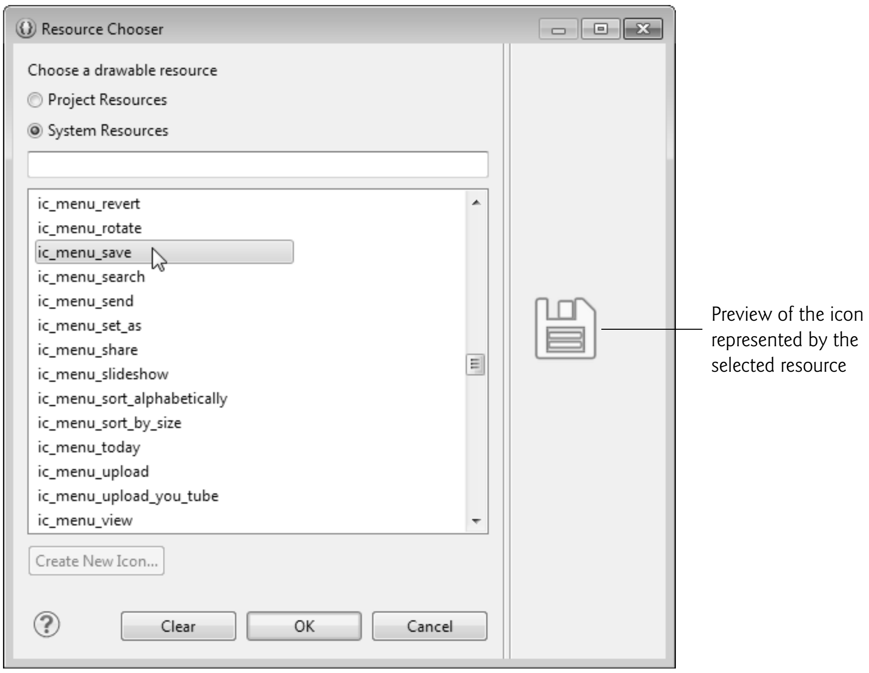


Fig. 4.13 | **Resource Chooser** dialog.

With the `tagEditText` selected, remove the **Ems** property's value from the **TextView** node in the **Properties** window. Then set the following properties:

- **Width** (**Layout Parameters** node): `0dp`—The IDE recommends this value when you also set the **Weight** property, so that the IDE can lay out the components more efficiently.

- **Height** (**Layout Parameters** node): `wrap_content`
- **Gravity** (**Layout Parameters** node): `bottom|fill_horizontal`—This aligns the bottom of the `tagEditText` with the bottom of the `saveButton` and indicates that `tagEditText` should fill the available horizontal space.
- **Weight** (**Layout Parameters** node): `1`—This makes the `tagEditText` more importance than the `saveButton` in this row. When Android lays out the row, the `saveButton` will occupy only the space it needs and the `tagEditText` will occupy all remaining horizontal space.
- **Hint**: `@string/tagPrompt`—Create a `String` resource with the value `"Tag your query"`.
- **IME Options** (**TextView** node): `actionDone`—This value indicates that `queryEditText's` keyboard will contain a **Done** button that the user can touch to dismiss the keyboard from the screen.

With the `saveButton` selected, clear the value of the **Weight** property (**Layout Parameters** node) then set the following properties:

- **Width** (**Layout Parameters** node): `wrap_content`
- **Height** (**Layout Parameters** node): `wrap_content`
- **Content Description**: `@string/saveDescription`—Create a string resource with the value `"Touch this button to save your tagged search"`.

Look-and-Feel Observation 4.3

Recall that it's considered a best practice in Android to ensure that every GUI component can be used with TalkBack. For components that don't have descriptive text, such as an `ImageButton`*, provide text for the component's* **Content Description** *property.*

Step 5: Creating the `GridLayout`'s Third Row

This row is a vertical `LinearLayout` containing a `TextView` and a `ListView`. Perform the following tasks to build the row's GUI:

1. Drag a **LinearLayout (Vertical)** component from the **Palette**'s **Layouts** section onto the `GridLayout` in the **Outline** window.
2. Drag a **Medium Text** component from the **Palette**'s **Form Widgets** section onto the `LinearLayout`. This creates a `TextView` that's preconfigured to display text in the theme's medium-sized text font.
3. Drag a **ListView** component from the **Palette**'s **Composite** section onto the `LinearLayout`, then set the **Id** property to `@android:id/list`—recall that this is the required **Id** for the `ListView` in a `ListActivity`'s custom layout.

With the vertical `LinearLayout` selected, set the following properties:

- **Height** (**Layout Parameters** node): `0dp`—The actual height is determined by the **Gravity** property.
- **Gravity** (**Layout Parameters** node): `fill`—This tells the `LinearLayout` to fill all available horizontal and vertical space.

- **Top** (located in the **Layout Parameters** node's **Margins** node): `@dimen/activity_vertical_margin`—This separates the top of the vertical `LinearLayout` from the horizontal `LinearLayout` in the GUI's second row.
- **Background** (**View** node): `@android:color/holo_blue_bright`—This is one of the predefined color resources in the app's Android theme.
- **Padding Left/Right** (**View** node): `@dimen/activity_horizontal_margin`—This ensures that the components in the vertical `LinearLayout` are inset by `16dp` from the left and right edges of the layout.
- **Padding Top** (**View** node): `@dimen/activity_vertical_margin`—This ensures that the top component within the vertical `LinearLayout` is inset by `16dp` from the top edge of the layout.

With the vertical `TextView` selected, set the following properties:

- **Width** (**Layout Parameters** node): `match_parent`
- **Height** (**Layout Parameters** node): `wrap_content`
- **Gravity** (**Layout Parameters** node): `fill_horizontal`—This makes the `TextView` fill the width of the vertical `LinearLayout` (minus the *padding* in the layout).
- **Gravity** (**TextView** node): `center_horizontal`—This centers the `TextView`'s text.
- **Text**: `@string/taggedSearches`—Create a string resource with the value `"Tagged Searches"`.
- **Padding Top** (**View** node): `@dimen/activity_vertical_margin`—This ensures that the top component within the vertical `LinearLayout` is inset by `16dp` from the top edge of the layout.

With the `ListView` selected, set the following properties:

- **Width** (**Layout Parameters** node): `match_parent`
- **Height** (**Layout Parameters** node): `0dp`—The IDE recommends this value when you also set the **Weight** property, so that the IDE can lay out the components more efficiently.
- **Weight** (**Layout Parameters** node): `1`
- **Gravity** (**Layout Parameters** node): `fill`—The `ListView` should fill all available horizontal and vertical space.
- **Padding Top** (**View** node): `@dimen/activity_vertical_margin`—This ensures that the top component within the vertical `LinearLayout` is inset by `16dp` from the top edge of the layout.
- **Top** and **Bottom** (located in the **Layout Parameters** node's **Margins** node): `@dimen/tagged_searches_padding`—Create a new `tagged_searches_padding` dimension resource by clicking the ellipsis button to the right of the **Top** property. In the **Resource Chooser** dialog, click **New Dimension…** to create a new dimension resource. Specify `tagged_searches_padding` for the **Name** and `8dp` for the **Value** and click **OK**, then select your new dimension resource and click **OK**. For the **Bottom** property, simply select this new dimension resource. These properties ensure that there is

an 8dp margin between the TextView and the top of the ListView and between the bottom of the ListView and the bottom of the vertical LinearLayout.

4.4.4 Graphical Layout Editor Toolbar

You've now completed the MainActivity's GUI. The **Graphical Layout** editor's toolbar (Fig. 4.14) contains various buttons that enable you to preview the design for other screen sizes and orientations. In particular, you can view thumbnail images of many screen sizes and orientations by clicking the down arrow next to the button and selecting either **Preview Representative Sample** or **Preview All Screen Sizes**. For each thumbnail, there are **+** and **–** buttons that you can click to zoom in and out. Figure 4.14 overviews some of the buttons in the **Graphical Layout** editor's toolbar.

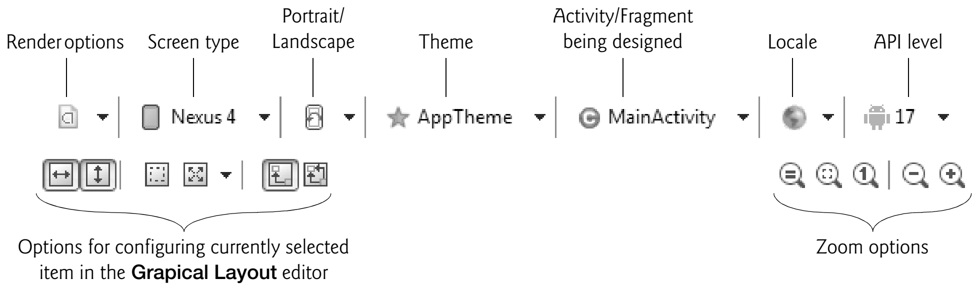

Render options
Screen type
Portrait/ Landscape
Theme
Activity/Fragment being designed
Locale
API level
Nexus 4
AppTheme
MainActivity
17
Options for configuring currently selected item in the **Grapical Layout** editor
Zoom options

Fig. 4.14 | Canvas configuration options.

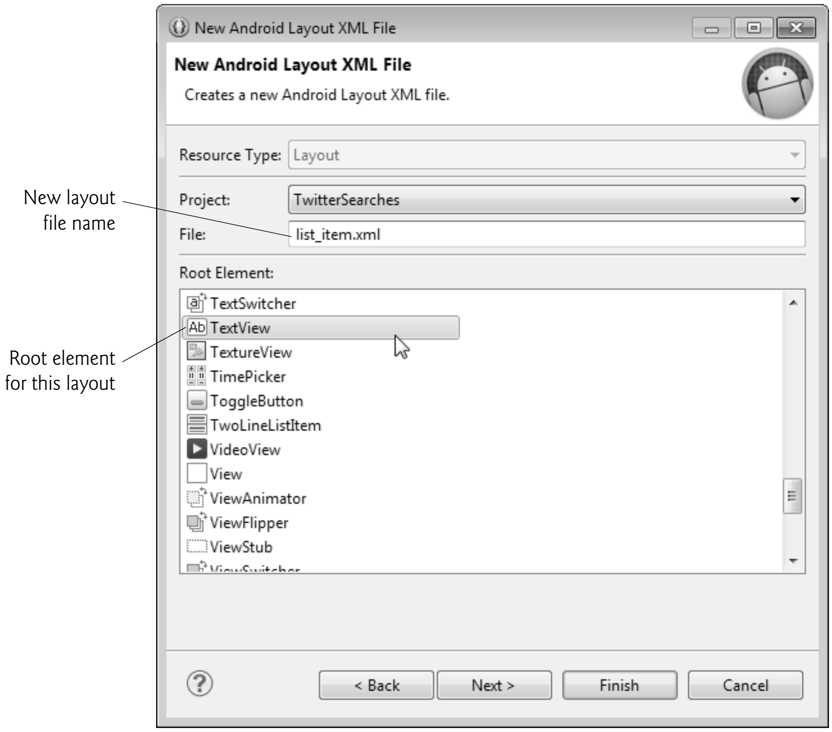
Option	Description
Render options	View one design screen at a time or see your design on a variety of screen sizes all at once.
Screen type	Android runs on a wide variety of devices, so the **Graphical Layout** editor comes with many device configurations that represent various screen sizes and resolutions that you can use to design your GUI. In this book, we use the predefined **Nexus 4**, **Nexus 7** and **Nexus 10** screens, depending on the app. In Fig. 4.14, we selected **Nexus 4**.
Portrait/Landscape	Toggles the design area between *portrait* and *landscape* orientations.
Theme	Can be used to set the theme for the GUI.
Activity/Fragment being designed	Shows the Activity or Fragment class that corresponds to the GUI being designed.
Locale	For *internationalized* apps (Section 2.8), allows you to select a specific localization, so that you can see, for example, what your design looks like with different language strings.
API level	Specifies the target API level for the design. With each new API level, there have typically been new GUI features. The **Graphical Layout** editor window shows only features that are available in the selected API level.

Fig. 4.15 | Explanation of the canvas configuration options.

4.4.5 `ListView` Item's Layout: `list_item.xml`

When populating a `ListView` with data, you must specify the format that's applied to each list item. Each list item in this app displays the `String` tag name for one saved search. To specify each list item's formatting, you'll create a new layout that contains only a `TextView` with the appropriate formatting. Perform the following steps:

1. In the **Package Explorer** window, expand the project's `res` folder, then right click the `layout` folder and select **New > Other...** to display the **New** dialog.
2. In the **Android** node, select **Android XML Layout File** and click **Next >** to display the dialog in Fig. 4.16, then configure the file as shown. The new layout's file name is `list_item.xml` and the root element in the layout is a `TextView`.
3. Click **Finish** to create the file.



Resource Type: Layout
Project: TwitterSearches
File: list_item.xml

Fig. 4.16 | Creating a new `list_item.xml` layout in the **New Android Layout XML File** dialog.

The IDE opens the new layout in the **Graphical Layout** editor. Select the `TextView` in the **Outline** window, then set the following properties:

- **Id**: `@+id/textView`—GUI component **Id**s begin with a lowercase first letter by convention.
- **Height** (**Layout Parameters** node): `?android:attr/listPreferredItemHeight`—This value is a predefined Android resource that represents a list item's preferred

height for responding properly to user touches with a minimal chance of touching the wrong item.

Look-and-Feel Observation 4.4

The Android design guidelines specify that the minimum recommended size for a touchable item on the screen is `48dp-by-48dp`. *For more information on GUI sizing and spacing, see* `http://developer.android.com/design/style/metrics-grids.html`.

- **Gravity** (**Layout Parameters** node): `center_vertical`—The `TextView` should be centered vertically within the `ListView` item.
- **Text Appearance** (**TextView** node): `?android:attr/textAppearanceMedium`—This is the predefined theme resource that specifies the font size for medium-sized text.

List Items That Display Multiple Pieces of Data

If a list item should display multiple pieces of data, you'll need a list-item layout that consists of multiple elements, and each element will need an `android:id` attribute.

Other Predefined Android Resources

There are many predefined Android resources like the ones used to set the **Height** and **Text Appearance** for a list item. You can view the complete list at:

```
http://developer.android.com/reference/android/R.attr.html
```

To use a value in your layouts, specify it in the format

```
?android:attr/resourceName
```

4.5 Building the `MainActivity` Class

Figures 4.17–4.27 implement the **Twitter Searches** app's logic in the class `MainActivity`, which extends `ListActivity`. The default code for class `MainActivity` included an `onCreateOptionsMenu` method, which we removed because it's not used in this app—we'll discuss `onCreateOptionsMenu` in Chapter 5. Throughout this section, we assume that you create the necessary `String` resources as you encounter them in the code descriptions.

4.5.1 package and `import` Statements

Figure 4.17 shows the app's `package` and `import` statements. The `package` statement (inserted in line 4 by the IDE when you created the project) indicates that the class in this file is part of the `com.deitel.twittersearches` package. Lines 6–26 import the classes and interfaces the app uses.

```
1  // MainActivity.java
2  // Manages your favorite Twitter searches for easy
3  // access and display in the device's web browser
4  package com.deitel.twittersearches;
5
```

Fig. 4.17 | `MainActivity`'s `package` and `import` statements. (Part 1 of 2.)

```
 6  import java.util.ArrayList;
 7  import java.util.Collections;
 8  
 9  import android.app.AlertDialog;
10  import android.app.ListActivity;
11  import android.content.Context;
12  import android.content.DialogInterface;
13  import android.content.Intent;
14  import android.content.SharedPreferences;
15  import android.net.Uri;
16  import android.os.Bundle;
17  import android.view.View;
18  import android.view.View.OnClickListener;
19  import android.view.inputmethod.InputMethodManager;
20  import android.widget.AdapterView;
21  import android.widget.AdapterView.OnItemClickListener;
22  import android.widget.AdapterView.OnItemLongClickListener;
23  import android.widget.ArrayAdapter;
24  import android.widget.EditText;
25  import android.widget.ImageButton;
26  import android.widget.TextView;
27  
```

Fig. 4.17 | MainActivity's package and import statements. (Part 2 of 2.)

Lines 6–7 import the ArrayList and Collections classes from the java.util package. We use class ArrayList to maintain the list of tags for the saved searches, and class Collections to *sort* the tags so they appear in alphabetical order. Of the remaining import statements, we consider only those for the features introduced in this chapter:

- Class AlertDialog of package android.app (line 9) is used to display dialogs.
- Class ListActivity of package android.app (line 10) is MainActivity's superclass, which provides the app's ListView and methods for manipulating it.
- Class **Context** of package **android.content** (line 11) provides access to information about the environment in which the app is running and allows you to use various Android services. We'll be using a constant from this class when we programmatically hide the soft keyboard after the user saves a search.
- Class **DialogInterface** of package android.content (line 12) contains the nested interface **OnClickListener**. We implement this interface to handle the events that occur when the user touches a button on an AlertDialog.
- Class Intent of package android.content (line 13) is used to create an object that specifies an *action* to be performed and the *data* to be acted upon—Android uses Intents to launch the appropriate activities. We'll use this class to launch the device's web browser to display Twitter search results and to display an *intent chooser* so the user can choose how to share a search.
- Class SharedPreferences of package android.content (line 14) is used to manipulate *persistent key–value pairs* that are stored in files associated with the app.
- Class **Uri** of package **android.net** (line 15) enables us to convert a URL into the format required by an Intent that launches the device's web browser.

- Class **View** of package **android.view** (line 17) is used in various event-handling methods to represent the GUI component that the user interacted with to initiate an event.
- Class View contains the nested interface **OnClickListener** (line 18). We implement this interface to handle the event raised when the user touches the ImageButton for saving a search.
- Class **InputMethodManager** of package **android.view.inputmethod** (line 19) enables us to hide the soft keyboard when the user saves a search.
- Package android.widget (lines 20–26) contains the GUI components and layouts that are used in Android GUIs. Class **AdapterView** (line 20) is the base class of ListView and is used when setting up the ListView's adapter (which supplies the ListView's items). You implement interface **AdapterView.OnItemClickListener** (line 21) to respond when the user *touches* an item in a ListView. You implement interface **AdapterView.OnItemLongClickListener** (line 22) to respond when the user *long presses* an item in a ListView. Class **ArrayAdapter** (line 23) is used to *bind* items to a ListView. Class ImageButton (line 25) represents a button that displays an image.

4.5.2 Extending ListActivity

MainActivity (Figs. 4.18–4.27) is the **Twitter Searches** app's only Activity class. When you created the TwitterSearches project, the IDE generated MainActivity as a subclass of Activity and provided the shell of an overridden onCreate method, which every Activity subclass *must* override. We changed the superclass to ListActivity (Fig. 4.18, line 28). When you make this change, the IDE does not recognize class ListActivity, so you must update your import statements. In the IDE, you can use **Source > Organize Imports** to update the import statements. Eclipse underlines any class or interface name that it does not recognize. In this case, if you hover the mouse over the class or interface name, a list of *quick fixes* will be displayed. If the IDE recognizes the name, it will suggest the missing import statement you need to add—simply click the name to add it.

```
28   public class MainActivity extends ListActivity
29   {
```

Fig. 4.18 | Class MainActivity is a subclass of ListActivity.

4.5.3 Fields of Class MainActivity

Figure 4.19 contains class MainActivity's static and instance variables. The String constant SEARCHES (line 31) represents the name of the file that will store the searches on the device. Lines 33–34 declare EditTexts that we'll use to access the queries and tags that the user enters. Line 35 declares the SharedPreferences instance variable savedSearches, which will be used to manipulate the *key–value pairs* representing the user's saved searches. Line 36 declares the ArrayList<String> that will store the sorted tag names for the user's searches. Line 37 declares the ArrayAdapter<String> that uses the contents of the ArrayList<String> as the source of the items displayed in MainActivity's ListView.

Good Programming Practice 4.1

For readability and modifiability, use `String` constants to represent filenames (and other `String` literals) that do not need to be localized, and thus are not defined in `strings.xml`.

```java
   // name of SharedPreferences XML file that stores the saved searches
   private static final String SEARCHES = "searches";

   private EditText queryEditText; // EditText where user enters a query
   private EditText tagEditText; // EditText where user tags a query
   private SharedPreferences savedSearches; // user's favorite searches
   private ArrayList<String> tags; // list of tags for saved searches
   private ArrayAdapter<String> adapter; // binds tags to ListView

```

Fig. 4.19 | Fields of class `MainActivity`.

4.5.4 Overriding `Activity` Method onCreate

The `onCreate` method (Fig. 4.20) is called by the system:

- when the app *loads*
- if the app's process was *killed* by the operating system while the app was in the background, and the app is then *restored*
- each time the configuration changes, such as when the user *rotates* the device or *opens* or *closes* a physical keyboard.

The method initializes the `Activity`'s instance variables and GUI components—we keep it simple so the app loads quickly. Line 43 makes the *required* call to the superclass's `onCreate` method. As in the previous app, the call to `setContentView` (line 44) passes the constant `R.layout.activity_main` to *inflate the GUI* from `activity_main.xml`.

```java
39    // called when MainActivity is first created
40    @Override
41    protected void onCreate(Bundle savedInstanceState)
42    {
43       super.onCreate(savedInstanceState);
44       setContentView(R.layout.activity_main);
45
46       // get references to the EditTexts
47       queryEditText = (EditText) findViewById(R.id.queryEditText);
48       tagEditText = (EditText) findViewById(R.id.tagEditText);
49
50       // get the SharedPreferences containing the user's saved searches
51       savedSearches = getSharedPreferences(SEARCHES, MODE_PRIVATE);
52
53       // store the saved tags in an ArrayList then sort them
54       tags = new ArrayList<String>(savedSearches.getAll().keySet());
55       Collections.sort(tags, String.CASE_INSENSITIVE_ORDER);
56
```

Fig. 4.20 | Overriding `Activity` method `onCreate`. (Part 1 of 2.)

```
      // create ArrayAdapter and use it to bind tags to the ListView
      adapter = new ArrayAdapter<String>(this, R.layout.list_item, tags);
      setListAdapter(adapter);

      // register listener to save a new or edited search
      ImageButton saveButton =
         (ImageButton) findViewById(R.id.saveButton);
      saveButton.setOnClickListener(saveButtonListener);

      // register listener that searches Twitter when user touches a tag
      getListView().setOnItemClickListener(itemClickListener);

      // set listener that allows user to delete or edit a search
      getListView().setOnItemLongClickListener(itemLongClickListener);
   } // end method onCreate

```

Fig. 4.20 | Overriding `Activity` method `onCreate`. (Part 2 of 2.)

Getting References to the ***`EditText`s***
Lines 47–48 obtain references to the `queryEditText` and `tagEditText` to initialize the corresponding instance variables.

Getting a ***`SharedPreferences`*** *Object*
Line 51 uses the method **`getSharedPreferences`** (inherited from class `Context`) to get a `SharedPreferences` object that can read existing *tag–query pairs* (if any) from the `SEARCHES` file. The first argument indicates the name of the file that contains the data. The second argument specifies the accessibility of the file and can be set to one of the following options:

- **`MODE_PRIVATE`**—The file is accessible *only* to this app. In most cases, you'll use this option.
- **`MODE_WORLD_READABLE`**—Any app on the device can *read* from the file.
- **`MODE_WORLD_WRITABLE`**—Any app on the device can *write* to the file.

These constants can be combined with the bitwise OR operator (`|`). We aren't reading a lot of data in this app, so it's fast enough to load the searches in `onCreate`.

Performance Tip 4.1
Lengthy data access should not be done in the UI thread; otherwise, the app will display an Application Not Responding (ANR) dialog—typically after five seconds of preventing the user from interacting with the app. For information on designing responsive apps, see `http://developer.android.com/guide/practices/design/responsiveness.html`.

Getting the Keys Stored in the ***`SharedPreferences`*** *Object*
We'd like to display the search tags alphabetically so the user can easily find a search to perform. First, line 54 gets the `Strings` representing the keys in the `SharedPreferences` object and stores them in `tags` (an `ArrayList<String>`). `SharedPreferences` method **`getAll`** returns all the saved searches as a `Map` (package `java.util`)—a collection of key–value pairs. We then call method **`keySet`** on that object to get all the keys as a `Set` (package `java.util`)—a collection of unique values. The result is used to initialize `tags`.

Sorting the ***`ArrayList`*** *of Tags*
Line 55 uses **`Collections.sort`** to sort `tags`. Since the user could enter tags using mixtures of uppercase and lowercase letters, we chose to perform a *case-insensitive sort* by passing the predefined `Comparator<String>` object **`String.CASE_INSENSITIVE_ORDER`** as the second argument to `Collections.sort`.

Using an ***`ArrayAdapter`*** *to Populate the* ***`ListView`***
To display the results in a `ListView` we create a new `ArrayAdapter<String>` object (line 58) which maps the contents `tags` to `TextView`s that are displayed in `MainActivity`'s `ListView`. The `ArrayAdapter<String>`'s constructor receives:

- the `Context` (`this`) in which the `ListView` is displayed—`this` is the `MainActivity`
- the resource ID (`R.layout.list_item`) of the layout that's used to display each item in the `ListView`
- a `List<String>` containing the items to display—`tags` is an `ArrayList<String>`, which implements interface `List<String>`, so `tags` is a `List<String>`.

Line 59 uses inherited `ListActivity` method **`setListAdapter`** to bind the `ListView` to the `ArrayAdapter`, so that the `ListView` can display the data.

Registering Listeners for the ***`saveButton`*** *and* ***`ListView`***
Lines 62–63 obtain a reference to the `saveButton` and line 64 registers its listener—instance variable `saveButtonListener` refers to an *anonymous-inner-class object* that implements interface `OnClickListener` (Fig. 4.21). Line 67 uses inherited `ListActivity` method **`getListView`** to get a reference to this activity's `ListView`, then registers the `ListView`'s `OnItemClickListener`—instance variable `itemClickListener` refers to an *anonymous inner class object* that implements this interface (Fig. 4.24). Similarly, line 70 registers the `ListView`'s `OnItemLongClickListener`—instance variable `itemLongClickListener` refers to an *anonymous-inner-class object* that implements this interface (Fig. 4.25).

4.5.5 Anonymous Inner Class That Implements the `saveButton`'s `OnClickListener` to Save a New or Updated Search

Figure 4.21 declares and initializes instance variable `saveButtonListener`, which refers to an *anonymous inner class object* that implements interface `OnClickListener`. Line 64 (Fig. 4.20) registered `saveButtonListener` as `saveButtons`'s event handler. Lines 76–109 override interface `OnClickListener`'s `onClick` method. If the user entered a query *and* a tag (lines 80–81), lines 83–84 call method `addTaggedSearch` (Fig. 4.23) to store the tag–query pair and lines 85–86 clear the two `EditText`s. Lines 88–90 hide the soft keyboard.

```
73     // saveButtonListener saves a tag-query pair into SharedPreferences
74     public OnClickListener saveButtonListener = new OnClickListener()
75     {
76        @Override
77        public void onClick(View v)
78        {
```

Fig. 4.21 | Anonymous inner class that implements the `saveButton`'s `OnClickListener` to save a new or updated search. (Part 1 of 2.)

```
 79            // create tag if neither queryEditText nor tagEditText is empty
 80            if (queryEditText.getText().length() > 0 &&
 81               tagEditText.getText().length() > 0)
 82            {
 83               addTaggedSearch(queryEditText.getText().toString(),
 84                  tagEditText.getText().toString());
 85               queryEditText.setText(""); // clear queryEditText
 86               tagEditText.setText(""); // clear tagEditText
 87
 88               ((InputMethodManager) getSystemService(
 89                  Context.INPUT_METHOD_SERVICE)).hideSoftInputFromWindow(
 90                  tagEditText.getWindowToken(), 0);
 91            }
 92            else // display message asking user to provide a query and a tag
 93            {
 94               // create a new AlertDialog Builder
 95               AlertDialog.Builder builder =
 96                  new AlertDialog.Builder(MainActivity.this);
 97
 98               // set dialog's title and message to display
 99               builder.setMessage(R.string.missingMessage);
100
101               // provide an OK button that simply dismisses the dialog
102               builder.setPositiveButton(R.string.OK, null);
103
104               // create AlertDialog from the AlertDialog.Builder
105               AlertDialog errorDialog = builder.create();
106               errorDialog.show(); // display the modal dialog
107            }
108         } // end method onClick
109      }; // end OnClickListener anonymous inner class
110
```

Fig. 4.21 | Anonymous inner class that implements the `saveButton`'s `OnClickListener` to save a new or updated search. (Part 2 of 2.)

Configuring an `AlertDialog`

If the user did not enter a query *and* a tag, lines 92–108 display an `AlertDialog` indicating that the user must enter both. An `AlertDialog.Builder` object (lines 95–96) helps you configure and create an `AlertDialog`. The argument to the constructor is the `Context` in which the dialog will be displayed—in this case, the `MainActivity`, which we refer to via its `this` reference. To access `this` from an *anonymous inner class*, you must fully qualify `this` with the outer class's name. Line 99 sets the dialog's message with the `String` resource `R.string.missingMessage` ("`Enter both a Twitter search query and a tag`").

Look-and-Feel Observation 4.5

You can set an `AlertDialog`*'s title (which appears above the dialog's message) with* `AlertDialog.Builder` *method* `setTitle`*. According to the Android design guidelines for dialogs (*`http://developer.android.com/design/building-blocks/dialogs.html`*), most dialogs do not need titles. A dialog should display a title for "a high-risk operation involving potential loss of data, connectivity, extra charges, and so on." Also, dialogs that display lists of options use the title to specify the dialog's purpose.*

Adding **`String`** *Resources to* **`strings.xml`**

To create `String` resources like `R.string.missingMessage`, open the `strings.xml` file located in the project's `res/values` folder. The IDE shows this file in a *resource editor* that has two tabs—**Resources** and **strings.xml**. In the **Resources** tab, you can click **Add...** to display the dialog in Fig. 4.22. Selecting **String** and clicking **OK** displays **Name** and **Value** textfields where you can enter a new `String` resource's name (e.g., `missingMessage`) and value. Save your `strings.xml` file after making changes. You can also use the resource editor's **Resource** tab to select an existing `String` resource to change its name and value.

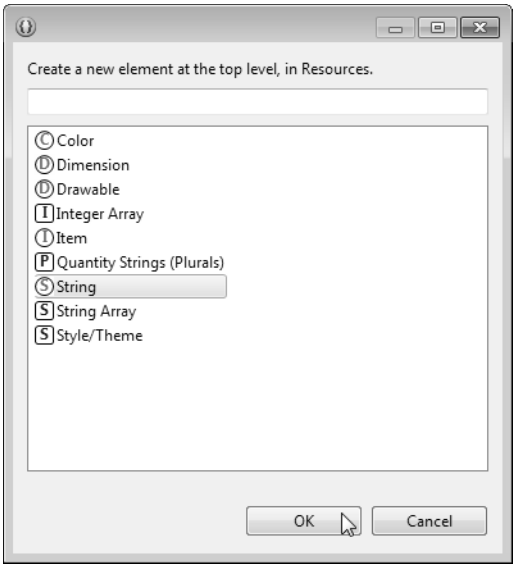

Create a new element at the top level, in Resources.
Color
Dimension
Drawable
Integer Array
Item
Quantity Strings (Plurals)
String
String Array
Style/Theme
OK
Cancel

Fig. 4.22 | Adding a `String` resource.

Specifying the **`AlertDialog`***'s Positive Button*

In this `AlertDialog`, we need only one button that allows the user to acknowledge the message. We specify this as the dialog's *positive* button (Fig. 4.21, line 102)—touching this button indicates that the user acknowledges the message displayed in the dialog. Method `setPositiveButton` receives the button's label (specified with the `String` resource `R.string.OK`) and a reference to the button's event handler. For this dialog, we don't need to respond to the event, so we specify `null` for the event handler. When the user touches the button, the dialog is simply dismissed from the screen.

Creating and Showing the **`AlertDialog`**

You create the `AlertDialog` by calling the `AlertDialog.Builder`'s `create` method (line 105) and display the *modal dialog* by calling `AlertDialog`'s `show` method (line 106).

4.5.6 addTaggedSearch Method

The event handler in Fig. 4.21 calls `MainActivity` method `addTaggedSearch` (Fig. 4.23) to add a new search to `savedSearches` or to modify an existing search.

```
111      // add new search to the save file, then refresh all Buttons
112      private void addTaggedSearch(String query, String tag)
113      {
114         // get a SharedPreferences.Editor to store new tag/query pair
115         SharedPreferences.Editor preferencesEditor = savedSearches.edit();
116         preferencesEditor.putString(tag, query); // store current search
117         preferencesEditor.apply(); // store the updated preferences
118
119         // if tag is new, add to and sort tags, then display updated list
120         if (!tags.contains(tag))
121         {
122            tags.add(tag); // add new tag
123            Collections.sort(tags, String.CASE_INSENSITIVE_ORDER);
124            adapter.notifyDataSetChanged(); // rebind tags to ListView
125         }
126      }
127
```

Fig. 4.23 | `addTaggedSearch` method of class `MainActivity`.

Editing a **`SharedPreferences`** *Object's Contents*

To change a `SharedPreferences` object's contents, you must first call its **`edit`** method to obtain a `SharedPreferences.Editor` object (line 115), which can add key–value pairs to, remove key–value pairs from, and modify the value associated with a particular key in a `SharedPreferences` file. Line 116 calls `SharedPreferences.Editor` method **`putString`** to save the search's tag (the key) and query (the corresponding value)—if the tag already exists in the `SharedPreferences` this updates the value. Line 117 *commits* the changes by calling `SharedPreferences.Editor` method **`apply`** to make the changes to the file.

Notifying the **`ArrayAdapter`** *That Its Data Has Changed*

When the user adds a new search, the `ListView` should be updated to display it. Lines 120–125 determine whether a new tag was added. If so, lines 122–123 add the new search's tag to `tags`, then sort `tags`. Line 124 calls the `ArrayAdapter`'s **`notifyDataSetChanged`** method to indicate that the underlying data in tags has changed. The adapter then notifies the `ListView` to update its list of displayed items.

4.5.7 Anonymous Inner Class That Implements the `ListView`'s `OnItemClickListener` to Display Search Results

Figure 4.24 declares and initializes instance variable `itemClickListener`, which refers to an *anonymous inner-class object* that implements interface `OnItemClickListener`. Line 67 (Fig. 4.20) registered `itemClickListener` as the `ListView`'s event handler that responds when the user *touches* an item in the `ListView`. Lines 131–145 override interface `OnItemClickListener`'s `onItemClick` method. The method's arguments are:

- The `AdapterView` where the user touched an item. The `?` in `AdapterView<?>` is a *wildcard* in Java generics indicating method `onItemClick` can receive an `AdapterView` that displays *any* type of data—in this case, a `ListView<String>`.
- The `View` that the user touched in the `AdapterView`—in this case, the `TextView` that displays a search tag.

- The *zero-based* index number of the item the user touched.
- The row ID of the item that was touched—this is used primarily for data obtained from a database (as you'll do in Chapter 8).

```
128    // itemClickListener launches web browser to display search results
129    OnItemClickListener itemClickListener = new OnItemClickListener()
130    {
131       @Override
132       public void onItemClick(AdapterView<?> parent, View view,
133          int position, long id)
134       {
135          // get query string and create a URL representing the search
136          String tag = ((TextView) view).getText().toString();
137          String urlString = getString(R.string.searchURL) +
138             Uri.encode(savedSearches.getString(tag, ""), "UTF-8");
139
140          // create an Intent to launch a web browser
141          Intent webIntent = new Intent(Intent.ACTION_VIEW,
142             Uri.parse(urlString));
143
144          startActivity(webIntent); // launches web browser to view results
145       }
146    }; // end itemClickListener declaration
147
```

Fig. 4.24 | Anonymous inner class that implements the `ListView`'s `OnItemClickListener` to display search results.

Getting `String` *Resources*

Line 136 gets the text of the `View` that the user touched in the `ListView`. Lines 137–138 create a `String` containing the Twitter search URL and the query to perform. First, line 137 calls `Activity`'s inherited method **`getString`** with one argument to get the `String` resource named `searchURL`, which contains the Twitter search page's URL:

```
http://mobile.twitter.com/search/
```

As with all the `String` resources in this app, you should add this resource to `strings.xml`.

Getting `String`*s from a* `SharedPreferences` *Object*

We append the result of line 138 to the search URL to complete the `urlString`. `SharedPreferences` method **`getString`** returns the query associated with the `tag`. If the `tag` does not already exist, the second argument (`""` in this case) is returned. Line 138 passes the query to `Uri` method `encode`, which *escapes* any special URL characters (such as `?`, `/`, `:`, etc.) and returns a so-called *URL-encoded* `String`. This is important to ensure that the Twitter web server that receives the request can parse the URL properly to obtain the search query.

Creating an `Intent` *to Launch the Device's Web Browser*

Lines 141–142 create a new `Intent`, which we'll use to *launch* the device's *web browser* and display the search results. `Intent`s can be used to launch other activities in the same app or in other apps. The first argument of `Intent`'s constructor is a constant describing the *action* to perform. **Intent.ACTION_VIEW** indicates that we'd like to display a representation of the data. Many constants are defined in the Intent class describing actions such as *searching*, *choosing*, *sending* and *playing*. The second argument (line 142) is a **Uri** (uniform resource identifier) representing the *data* on which we want to perform the action. Class Uri's **parse method** converts a String representing a URL (uniform resource locator) to a Uri.

*Starting an **Activity** for an **Intent***

Line 144 passes the Intent to the inherited Activity method startActivity, which starts an Activity that can perform the specified *action* on the given *data*. In this case, because we've specified to view a URI, the Intent launches the device's web browser to display the corresponding web page. This page shows the results of the supplied Twitter search.

4.5.8 Anonymous Inner Class That Implements the ListView's OnItemLongClickListener to Share, Edit or Delete a Search

Figure 4.25 declares and initializes instance variable itemLongClickListener, which refers to an *anonymous inner-class object* that implements interface OnItemLongClickListener. Line 70 (Fig. 4.20) registered itemLongClickListener as the ListView's event handler that responds when the user long presses an item in the ListView. Lines 153–210 override interface OnItemLongClickListener's onItemLongClick method.

```
148    // itemLongClickListener displays a dialog allowing the user to delete
149    // or edit a saved search
150    OnItemLongClickListener itemLongClickListener =
151       new OnItemLongClickListener()
152       {
153          @Override
154          public boolean onItemLongClick(AdapterView<?> parent, View view,
155             int position, long id)
156          {
157             // get the tag that the user long touched
158             final String tag = ((TextView) view).getText().toString();
159
160             // create a new AlertDialog
161             AlertDialog.Builder builder =
162                new AlertDialog.Builder(MainActivity.this);
163
164             // set the AlertDialog's title
165             builder.setTitle(
166                getString(R.string.shareEditDeleteTitle, tag));
167
168             // set list of items to display in dialog
169             builder.setItems(R.array.dialog_items,
170                new DialogInterface.OnClickListener()
171                {
172                   // responds to user touch by sharing, editing or
173                   // deleting a saved search
```

Fig. 4.25 | Anonymous inner class that implements the ListView's OnItemLongClickListener to share, edit or delete. (Part 1 of 2.)

```
                  @Override
                  public void onClick(DialogInterface dialog, int which)
                  {
                     switch (which)
                     {
                        case 0: // share
                           shareSearch(tag);
                           break;
                        case 1: // edit
                           // set EditTexts to match chosen tag and query
                           tagEditText.setText(tag);
                           queryEditText.setText(
                              savedSearches.getString(tag, ""));
                           break;
                        case 2: // delete
                           deleteSearch(tag);
                           break;
                     }
                  }
               } // end DialogInterface.OnClickListener
            ); // end call to builder.setItems

            // set the AlertDialog's negative Button
            builder.setNegativeButton(getString(R.string.cancel),
               new DialogInterface.OnClickListener()
               {
                  // called when the "Cancel" Button is clicked
                  public void onClick(DialogInterface dialog, int id)
                  {
                     dialog.cancel(); // dismiss the AlertDialog
                  }
               }
            ); // end call to setNegativeButton

            builder.create().show(); // display the AlertDialog
            return true;
         } // end method onItemLongClick
      }; // end OnItemLongClickListener declaration

```

Fig. 4.25 | Anonymous inner class that implements the `ListView`'s `OnItemLongClickListener` to share, edit or delete. (Part 2 of 2.)

`final` Local Variables for Use in Anonymous Inner Classes

Line 158 gets the text of the item the user *long pressed* and assigns it to `final` local variable `tag`. Any local variable or method parameter that will be used in an anonymous inner class *must* be declared `final`.

`AlertDialog` That Displays a List of Items

Lines 161–166 create an `AlertDialog.Builder` and set the dialog's title to a formatted `String` in which `tag` replaces the format specifier in the resource `R.string.shareEditDeleteTitle` (which represents `"Share, Edit or Delete the search tagged as \"%s\""`).

Line 166 calls Activity's inherited method getString that receives *multiple arguments*—this first is a String resource ID representing a *format String* and the remaining arguments are the values that should replace the format specifiers in the format String. In addition to buttons, an AlertDialog can display a list of items in a ListView. Lines 169–194 specify that the dialog should display the array of Strings R.array.dialog_items (which represents the Strings "Share", "Edit" and "Delete") and define an anonymous inner class to respond when the user touches an item in the list.

Adding a **String** *Array Resource to* **strings.xml**

The array of Strings is defined as a String array resource in the strings.xml file. To add a String array resource to strings.xml:

1. Follow the steps in Section 4.5.5 to add a String resource, but select **String Array** rather than **String** in the dialog of Fig. 4.22, then click **OK**.
2. Specify the array's name (dialog_items) in the **Name** textfield.
3. Select the array in the list of resources at the left side of the resource editor.
4. Click Add... then click OK to add a new **Item** to the array.
5. Specify the new **Item**'s value in the **Value** textfield.

Perform these steps for the items Share, Edit and Delete (in that order), then save the strings.xml file.

Event Handler for the Dialog's List of Items

The anonymous inner class in lines 170–193 determines which item the user selected in the dialog's list and performs the appropriate action. If the user selects **Share**, shareSearch is called (line 180). If the user selects **Edit**, lines 184–186 display the search's query and tag in the EditTexts. If the user selects **Delete**, deleteSearch is called (line 189).

Configuring the Negative Button and Displaying the Dialog

Lines 197–206 configure the dialog's *negative* button to dismiss the dialog if the user decides not to share, edit or delete the search. Line 208 creates and shows the dialog.

4.5.9 shareSearch Method

Method shareSearch (Fig. 4.26) is called by the event handler in Fig. 4.25 when the user selects to share a search. Lines 217–218 create a String representing the search to share. Lines 221–227 create and configure an Intent that allows the user to send the search URL using an Activity that can handle the Intent.**ACTION_SEND** (line 222).

```
213      // allows user to choose an app for sharing a saved search's URL
214      private void shareSearch(String tag)
215      {
216         // create the URL representing the search
217         String urlString = getString(R.string.searchURL) +
218            Uri.encode(savedSearches.getString(tag, ""), "UTF-8");
219
```

Fig. 4.26 | shareSearch method of class MainActivity. (Part 1 of 2.)

```
220       // create Intent to share urlString
221       Intent shareIntent = new Intent();
222       shareIntent.setAction(Intent.ACTION_SEND);
223       shareIntent.putExtra(Intent.EXTRA_SUBJECT,
224          getString(R.string.shareSubject));
225       shareIntent.putExtra(Intent.EXTRA_TEXT,
226          getString(R.string.shareMessage, urlString));
227       shareIntent.setType("text/plain");
228
229       // display apps that can share text
230       startActivity(Intent.createChooser(shareIntent,
231          getString(R.string.shareSearch)));
232    }
233
```

Fig. 4.26 | `shareSearch` method of class `MainActivity`. (Part 2 of 2.)

Adding Extras to an `Intent`

An `Intent` includes a `Bundle` of *extras*—additional information that's passed to the `Activity` that handles the `Intent`. For example, an e-mail `Activity` can receive *extras* representing the e-mail's subject, CC and BCC addresses, and the body text. Lines 223–226 use `Intent` method **`putExtra`** to add an extra as a key–value pair to the `Intent`'s `Bundle`. The method's first argument is a `String` key representing the purpose of the extra and the second argument is the corresponding extra data. Extras may be primitive type values, primitive type arrays, entire `Bundle` objects and more—see class `Intent`'s documentation for a complete list of the `putExtra` overloads.

The extra at lines 223–224 specifies an e-mail's subject with the `String` resource `R.string.shareSubject` (`"Twitter search that might interest you"`). For an `Activity` that does *not* use a subject (such as sharing on a social network), this extra is *ignored*. The extra at lines 225–226 represents the text to share—a formatted `String` in which the `urlString` is substituted into the `String` resource `R.string.shareMessage` (`"Check out the results of this Twitter search: %s"`). Line 227 sets the `Intent`'s MIME type to `text/plain`—such data can be handled by any `Activity` capable of sending plain text messages.

Displaying an Intent Chooser

To display the *intent chooser* shown in Fig. 4.8(a), we pass the `Intent` and a `String` title to `Intent`'s `static` **`createChooser`** method (line 230). The resource `R.string.shareSearch` (`"Share Search to:"`) is used as the intent chooser's title. It's important to set this title to remind the user to select an appropriate `Activity`. You cannot control the apps installed on a user's phone or the `Intent` filters that can launch those apps, so it's possible that incompatible activities could appear in the chooser. Method `createChooser` returns an `Intent` that we pass to `startActivity` to display the intent chooser.

4.5.10 `deleteSearch` Method

The event handler in Fig. 4.25 calls method `deleteSearch` (Figure 4.27) when the user long presses a search tag and selects **Delete**. Before deleting the search, the app displays an `AlertDialog` to confirm the delete operation. Lines 241–242 set the dialog's title to a formatted `String` in which `tag` replaces the format specifier in the `String` resource

R.string.confirmMessage ("Are you sure you want to delete the search \"%s\"?"). Lines 245–254 configure the dialog's negative button to dismiss the dialog. The String resource R.string.cancel represents "Cancel". Lines 257–275 configure the dialog's positive button to remove the search. The String resource R.string.delete represents "Delete". Line 263 removes the tag from the tags collection, and lines 266–269 use a SharedPreferences.Editor to remove the search from the app's SharedPreferences. Line 272 then notifies the ArrayAdapter that the underlying data has changed so that the ListView can update its displayed list of items.

```
234    // deletes a search after the user confirms the delete operation
235    private void deleteSearch(final String tag)
236    {
237       // create a new AlertDialog
238       AlertDialog.Builder confirmBuilder = new AlertDialog.Builder(this);
239
240       // set the AlertDialog's message
241       confirmBuilder.setMessage(
242          getString(R.string.confirmMessage, tag));
243
244       // set the AlertDialog's negative Button
245       confirmBuilder.setNegativeButton(getString(R.string.cancel),
246          new DialogInterface.OnClickListener()
247          {
248             // called when "Cancel" Button is clicked
249             public void onClick(DialogInterface dialog, int id)
250             {
251                dialog.cancel(); // dismiss dialog
252             }
253          }
254       ); // end call to setNegativeButton
255
256       // set the AlertDialog's positive Button
257       confirmBuilder.setPositiveButton(getString(R.string.delete),
258          new DialogInterface.OnClickListener()
259          {
260             // called when "Cancel" Button is clicked
261             public void onClick(DialogInterface dialog, int id)
262             {
263                tags.remove(tag); // remove tag from tags
264
265                // get SharedPreferences.Editor to remove saved search
266                SharedPreferences.Editor preferencesEditor =
267                   savedSearches.edit();
268                preferencesEditor.remove(tag); // remove search
269                preferencesEditor.apply(); // saves the changes
270
271                // rebind tags ArrayList to ListView to show updated list
272                adapter.notifyDataSetChanged();
273             }
274          } // end OnClickListener
275       ); // end call to setPositiveButton
```

Fig. 4.27 | deleteSearch method of class MainActivity. (Part 1 of 2.)

```

        confirmBuilder.create().show(); // display AlertDialog
     } // end method deleteSearch
  } // end class MainActivity
```

Fig. 4.27 | `deleteSearch` method of class `MainActivity`. (Part 2 of 2.)

4.6 AndroidManifest.xml

In Section 3.6, you made two changes to the `AndroidManifest.xml` file:

- The first indicated that the **Tip Calculator** app supported only portrait orientation.
- The second forced the soft keyboard to be displayed when the app started executing so that the user could immediately enter a bill amount in the **Tip Calculator** app.

This app supports both portrait and landscape orientations. No changes are required to indicate this, because all apps support both orientations by default.

In this app, most users will launch this app so that they can perform one of their saved searches. When the first GUI component in the GUI is an `EditText`, Android gives that component the focus when the app loads. As you know, when an `EditText` receives the focus, its corresponding soft keyboard is displayed (unless a hardware keyboard is present). In this app, we want to prevent the soft keyboard from being displayed unless the user touches one of the app's `EditTexts`. To do so, follow the steps in Section 3.6 for setting the **Window soft input mode** option, but set its value to `stateAlwaysHidden`.

4.7 Wrap-Up

In this chapter, you created the **Twitter Searches** app. First you designed the GUI. We introduced the `ListView` component for displaying a scrollable list of items and used it to display the arbitrarily large list of saved searches. Each search was associated with an item in the `ListView` that the user could touch to pass the search to the device's web browser. You also learned how to create `String` resources for use in your Java code.

We stored the search tag–query pairs in a `SharedPreferences` file associated with the app and showed how to programmatically *hide* the soft keyboard. We also used a `SharedPreferences.Editor` object to store values in, modify values in and remove values from a `SharedPreferences` file. In response to the user touching a search tag, we loaded a `Uri` into the device's web browser by creating a new `Intent` and passing it to `Context`'s `startActivity` method. You also used an `Intent` to display an intent chooser allowing the user to select an Activity for sharing a search.

You used `AlertDialog.Builder` objects to configure and create `AlertDialogs` for displaying messages to the user. Finally, we discussed the `AndroidManifest.xml` file and showed you how to configure the app so that the soft keyboard is not displayed when the app is launched.

In Chapter 5, you'll build the **Flag Quiz** app in which the user is shown a graphic of a country's flag and must guess the country from 3, 6 or 9 choices. You'll use a menu and checkboxes to customize the quiz, limiting the flags and countries chosen to specific regions of the world.

Self-Review Exercises

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

a) __________ are typically used to launch activities—they indicate an action to be performed and the data on which that action is to be performed.

b) Class __________ of package `android.content` is used to manipulate *persistent key-value pairs* that are stored in files associated with the app.

c) Lengthy data access should never be done in the UI thread; otherwise, the app will display a(n) __________ dialog—typically after five seconds of inactivity.

d) An `Intent` is a description of an action to be performed with associated __________.

e) __________ `Intents` specify an exact `Activity` class to run in the same app.

f) When you create the project for each Android app in Eclipse, the ADT Plugin creates and configures the __________ file (also known as the app's manifest), which describes information about the app.

g) A standard `AlertDialog` may have up to three buttons that represent __________, __________ and __________ actions.

h) A(n) __________ creates an object that populates the `ListView` using data from an `ArrayList` collection object.

i) __________ is a subclass of `ImageView` which provides additional capabilities that enable an image to be used like a `Button` object.

4.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Extensive input/output should be performed on the UI thread; otherwise, this will affect your app's responsiveness.

b) Class `Context` of package `android.content` gives access to information about the app's environment and lets you use various Android services.

c) When an `Activity`'s primary task is to display a scrollable list of items, you can extend class `ListActivity`, which uses a `ListView` that occupies the entire app as its default layout.

d) `ListView` is a subclass of `Adapter`—a GUI component is bound to a data source.

e) When customizing a `ListActivity` subclass's GUI, the layout must contain a `ListView` with its **Id** attribute set to `"@android:id/list"`—the name that class `ListActivity` uses to reference its `ListView`.

Answers to Self-Review Exercises

4.1 a) `Intents`. b) `SharedPreferences`. c) Application Not Responding (ANR). d) data. e) Explicit. f) `AndroidManifest.xml`. g) positive, negative, neutral. h) `ArrayAdapter`. i) `ImageButton`.

4.2 a) False. Extensive input/output should *not* be performed on the UI thread, since that would affect your app's responsiveness. b) True. c) True. d) False. `ListView` is a subclass of `AdapterView`—a GUI component is bound to a data source via an `Adapter` object. e) True.

Exercises

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

a) A layout fills the entire client area of the screen if the layout's **Width** and **Height** properties (in the **Layout Parameters** section of the **Properties** window) are each set to __________.

b) __________ is used to display a scrollable list of views/items.

c) To create an object that populates the `ListView` using data from an `ArrayList` collection object, we use __________ of package `android.widget`.

d) `ImageButton` is a subclass of __________ which provides additional capabilities enabling an image to be used like a `Button` object.
e) If the Android system cannot find an activity to handle any intent action, then method `startActivity` throws an __________ `Exception`.
f) A(n) __________ is a GUI that shows a list of apps that can handle a specified Intent.

4.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) Android uses the intent messaging technique to communicate information between activities within one app or activities in separate apps.
b) You implement interface `View.OnClickListener` of package `android.view` to specify the code that should execute when the user touches a `Button`.
c) `SharedPreferences` objects can store keys and values of non-primitive data types.
d) An explicit `Intent` allows the system to launch the most appropriate `Activity` based on the type of data.

Project Exercises

4.3 *(Favorite Celebrities App)* Using the techniques you learned in this chapter, create a **Favorite Celebrities** app that allows a user to create a list of favorite celebrities.

4.4 *(Enhancements to the Favorite Celebrities App)* Enhance the **Favorite Celebrities** App to allow the user to search for celebrities based on filters like name, area of interest, location/country and popularity.

4.5 *(Flickr Searches App)* Investigate Flickr's photo search API (`http://www.flickr.com/services/api/flickr.photos.search.html`), then reimplement this chapter's **Twitter Searches** app as a **Flickr Searches** app.

4.6 *(Enhanced Flickr Searches App)* Enhance the **Flickr Searches** app from Exercise 4.5 to allow the user to add filters to searches (e.g., include only images containing a specific color, shape, object, etc.).

4.7 *(Word Scramble Game)* Create an app that scrambles the letters of a word or phrase and asks the user to enter the correct word or phrase. Keep track of the user's high score in the app's `SharedPreferences`. Include levels (three-, four-, five-, six- and seven-letter words). As a hint to the user, provide a definition with each word. [*Optional:* Locate a free dictionary web service, then use it to select the words and definitions.]

Advanced Project Exercises

4.8 *(Blackjack App)* Create a Blackjack card game app. Two cards each are dealt to the dealer and the player. (We provide card images with the book's examples.) The player's cards are dealt face up. Only the dealer's first card is dealt face up. Each card has a value. A card numbered 2 through 10 is worth its face value. Jacks, queens and kings each count as 10. Aces can count as 1 or 11—whichever value is more beneficial to the player. If the sum of the player's two initial cards is 21 (that is, the player was dealt a card valued at 10 and an ace, which counts as 11 in this situation), the player has "blackjack" and the dealer's face-down card is revealed. If the dealer does not have blackjack, the player immediately wins the game; otherwise, the hand is a "push" (that is, a tie) and no one wins the hand. If the player does not have blackjack, the player can begin taking additional cards one at a time. These cards are dealt face up, and the player decides when to stop taking cards. If the player "busts" (that is, the sum of the player's cards exceeds 21), the game is over, and the player loses. When the player stands (stops taking cards), the dealer's hidden card is revealed. If the dealer's total is 16 or less, the dealer must take another card; otherwise, the dealer must stay. The dealer must continue to take cards until the sum of the dealer's cards is greater than or equal to 17. If the dealer exceeds 21, the player wins. Otherwise, the hand with the higher point total wins. If the dealer and the player have the same point total, the game is a "push," and no one wins. The GUI for this app can be built using `ImageViews`, `TextViews` and `Buttons`.

4.9 *(Enhanced Blackjack App)* Enhance the Blackjack app in Exercise 4.8 as follows:

a) Provide a betting mechanism that allows the player to start with $1000 and adds or subtracts from that value based on whether the user wins or loses a hand. If the player wins with a non-blackjack hand, the bet amount is added to the total. If the player wins with blackjack, 1.5 times the bet amount is added to the total. If the player loses the hand, the bet amount is subtracted from the total. The game ends when the user runs out of money.
b) Locate images of casino chips and use them to represent the bet amount on the screen.
c) Investigate Blackjack rules online and provide capabilities for "doubling down," "surrendering" and other aspects of the game.
d) Some casinos use variations of the standard Blackjack rules. Provide options that allow the user to choose the rules under which the game should be played.
e) Some casinos use different numbers of decks of cards. Allow the user to choose how many decks should be used.
f) Allow the user to save the game's state to continue at a later time.

4.10 *(Other Card Game Apps)* Investigate the rules for any card game of your choice online and implement the game as an app.

4.11 *(Solitaire Card Game)* Search the web for the rules to various solitaire card games. Choose the version of the game you like then implement it. (We provide card images with the book's examples.)

5 Flag Quiz App

Objectives

In this chapter you'll:

- Use `Fragment`s to make better use of available screen real estate in an `Activity`'s GUI on phones and tablets.
- Display an options menu on the action bar to enable users to configure the app's preferences.
- Use a `Preference-Fragment` to automatically manage and persist an app's user preferences.
- Use an app's `assets` subfolders to organize image resources and manipulate them with an `AssetManager`.
- Define an animation and apply it to a `View`.
- Use a `Handler` to schedule a future task to perform on the GUI thread.
- Use `Toast`s to display messages briefly to the user.
- Launch a specific `Activity` with an explicit `Intent`.
- Use various collections from the `java.util` package.
- Define layouts for multiple device orientations.
- Use Android's logging mechanism to log error messages.

Outline

5.1 Introduction

The **Flag Quiz** app tests your ability to correctly identify 10 country flags (Fig. 5.1). Initially, the app presents a flag image and three guess `Button`s representing the possible country answers—one matches the flag and the others are randomly selected, nonduplicated incorrect answers. The app displays the user's progress throughout the quiz, showing the question number (out of 10) in a `TextView` above the current flag image. As you'll see, the app also allows you to control the quiz difficulty by specifying whether to display three, six or nine guess `Button`s, and by choosing the world regions that should be included in the quiz. These options are displayed differently based on the device that's running the app and the orientation of the device—the app supports portrait orientation on *any* device, but landscape orientation only on tablets. In portrait orientation, the app displays on the action bar an *options menu* containing a **Settings** menu item. When the user selects this item, the app

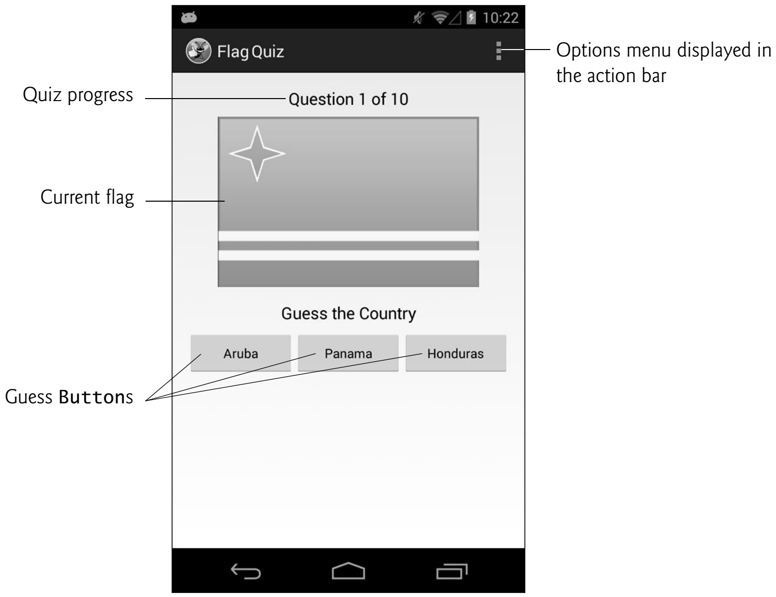


Fig. 5.1 | **Flag Quiz** app running on a smartphone in portrait orientation.

displays an `Activity` for setting the number of guess `Buttons` and the world regions to use in the quiz. On a tablet in landscape orientation (Fig. 5.2), the app uses a different layout that displays the app's settings at the left side of the screen and the quiz at the right side.

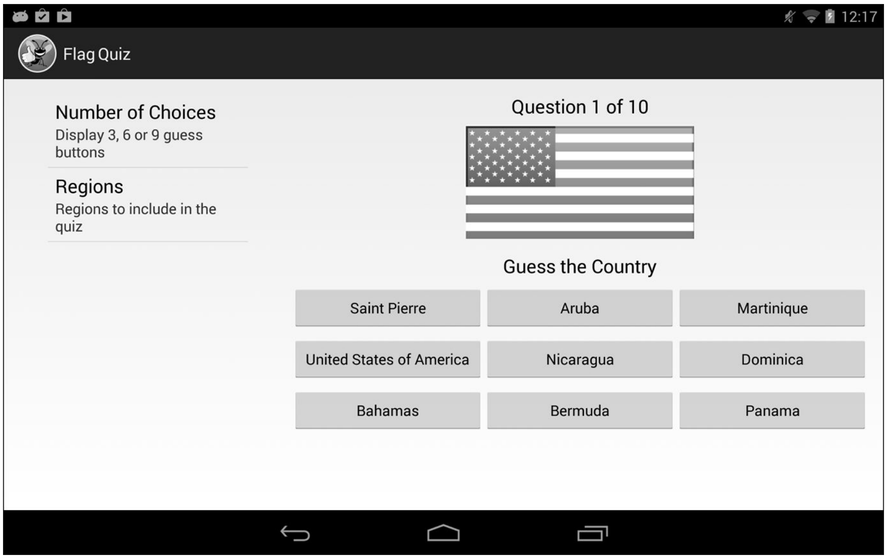


Fig. 5.2 | **Flag Quiz** app running on a tablet in landscape orientation.

First, you'll test-drive the app. Then we'll overview the technologies used to build it. Next, you'll design the app's GUI. Finally, we'll present the app's complete source code and walk through the code, discussing the app's new features in more detail.

5.2 Test-Driving the Flag Quiz App

You'll now test-drive the **Flag Quiz** app. Open the IDE and import the **Flag Quiz** app project. You can test this app on a phone AVD, tablet AVD or actual device. The screen captures in this chapter were taken on a Nexus 4 phone and a Nexus 7 tablet.

5.2.1 Importing the App and Running It

Perform the following steps to import the app into the IDE:

1. *Opening the* **Import** *dialog.* Select **File > Import...**.
2. *Importing the* **Flag Quiz** *app's project.* Expand the **General** node and select **Existing Projects into Workspace**. Click **Next >** to proceed to the **Import Projects** step. Ensure that **Select root directory** is selected, then click **Browse...**. Locate the `FlagQuiz` folder in the book's examples folder, select it and click **OK**. Ensure that **Copy projects into workspace** is *not* selected. Click **Finish** to import the project so that it appears in the **Package Explorer** window.
3. *Launching the* **Flag Quiz** *app.* Right click the `FlagQuiz` project and select **Run As > Android Application** to execute the app in the AVD or on a device. This builds the project and runs the app (Fig. 5.1 or Fig. 5.2).

5.2.2 Configuring the Quiz

When you first install and run the app, the quiz is configured to display three guess `Buttons` and to select flags from *all* of the world's regions. For this test-drive, you'll change the app's options to select flags only from North America and you'll keep the app's default setting of three guess `Buttons` per flag. On a phone, a tablet or AVD in portrait orientation, touch the *options menu* icon (⋮, Fig. 5.1) on the action bar to open the menu, then select **Settings** so you can view the app's options in the **Flag Quiz Settings** screen (Fig. 5.3(a)). On a tablet device or tablet AVD in *landscape* orientation, the app's settings options appear at the left side of the screen (Fig. 5.2). Touch **Number of Choices** to display the dialog (Fig. 5.3(b)) for selecting the number of `Buttons` that should be displayed with each flag. (On a tablet device or tablet AVD in landscape orientation, the entire app is grayed out and the dialog appears in the center of the screen.) By default, **3** is selected. To make the quiz more challenging, you can select **6** or **9** and touch **OK**; otherwise, touch **Cancel** to return to the **Flag Quiz Settings** screen. We used the default setting of three guess `Buttons` for this test-drive.

Next, touch **Regions** (Fig. 5.4(a)) to display the checkboxes representing the world regions (Fig. 5.4(b)). By default, all regions are enabled when the app is first executed, so any of the world's flags can be selected randomly for the quiz. Touch the checkboxes next to **Africa**, **Asia**, **Europe**, **Oceania** and **South America** to uncheck them—this excludes those regions' countries from the quiz. Touch **OK** to reset the quiz with the updated settings. On a phone, a tablet or AVD in portrait orientation, touch the back button (⮌) to return to the quiz. On a tablet device or tablet AVD in landscape orientation, a quiz with the updated settings is immediately displayed at the right side of the screen.

a) Menu with the user touching **Number of Choices** b) Dialog showing options for number of choices

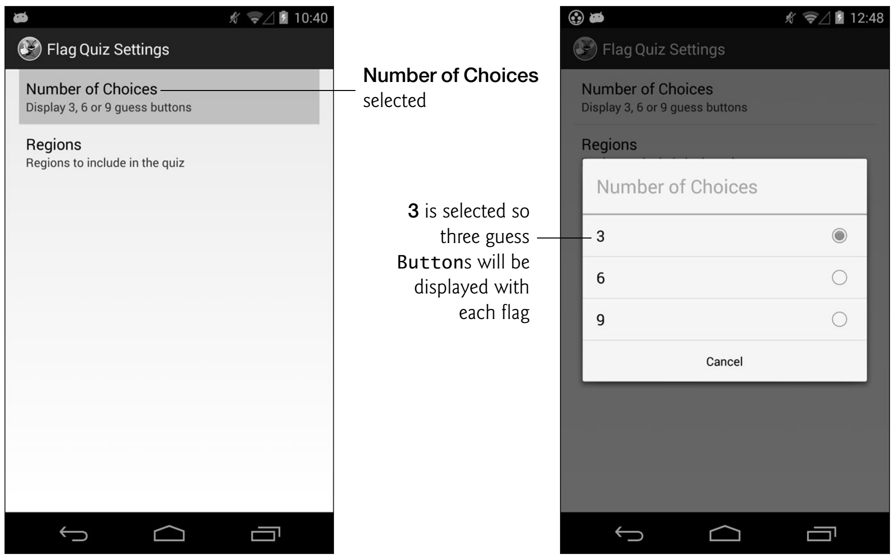

10:40
Flag Quiz Settings
Number of Choices
Display 3, 6 or 9 guess buttons
Regions
Regions to include in the quiz
Number of Choices selected
3 is selected so three guess `Buttons` will be displayed with each flag
12:48
Flag Quiz Settings
Number of Choices
3
6
9
Cancel

Fig. 5.3 | **Flag Quiz** settings screen and the **Number of Choices** dialog.

a) Menu with the user touching **Regions** b) Dialog showing regions

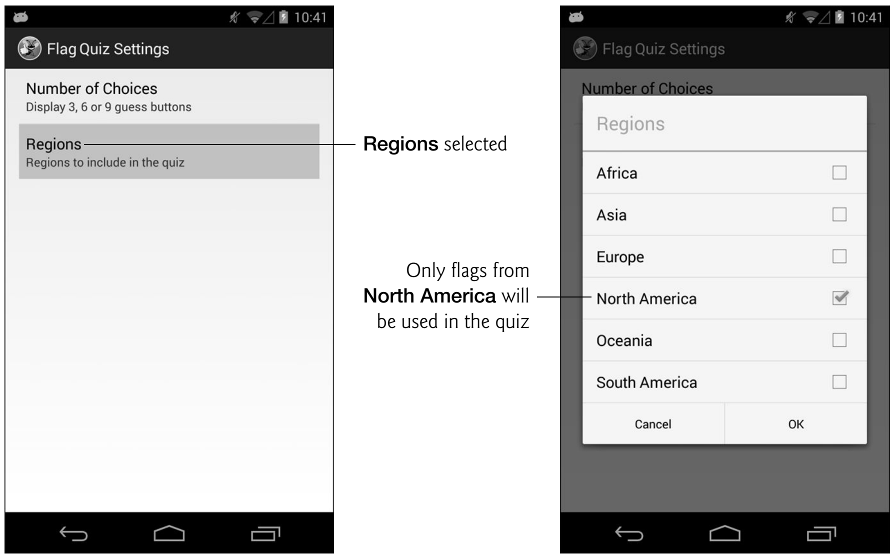


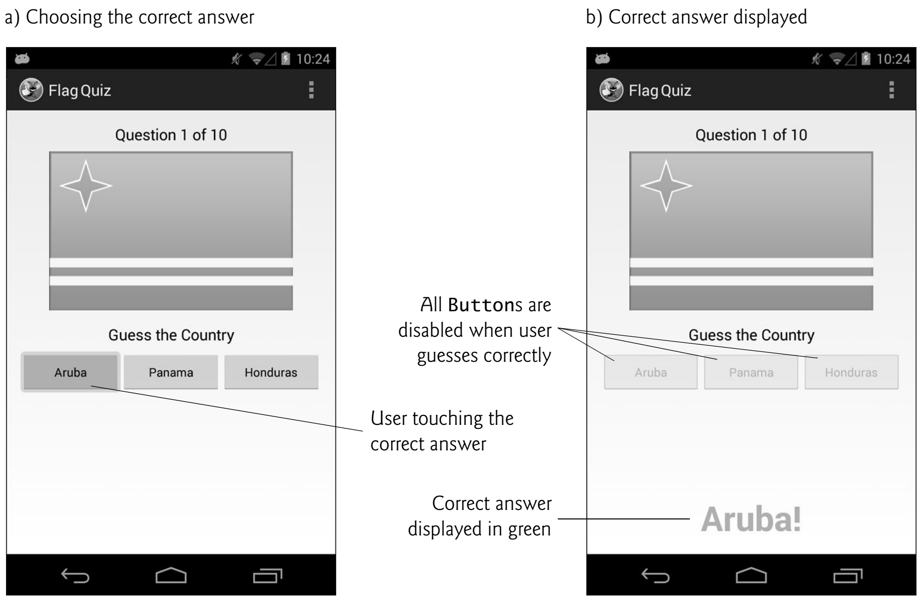
Fig. 5.4 | **Flag Quiz** settings screen and the **Regions** dialog.

5.2.3 Taking the Quiz

A new quiz starts with the number of answer choices you selected and flags from only the North America region. Work through the quiz by touching the guess `Button` for the country that you think matches each flag.

Making a Correct Selection

If the choice is correct (Fig. 5.5(a)), the app disables all the answer `Buttons` and displays the country name in green followed by an exclamation point at the bottom of the screen (Fig. 5.5(b)). After a short delay, the app loads the next flag and displays a new set of answer `Buttons`.



a) Choosing the correct answer
b) Correct answer displayed
10:24
Flag Quiz
Question 1 of 10
Guess the Country
Aruba
Panama
Honduras
All Buttons are disabled when user guesses correctly
User touching the correct answer
Correct answer displayed in green
Aruba!

Fig. 5.5 | User choosing the correct answer and the correct answer displayed.

User Making an Incorrect Selection

If you select incorrectly (Fig. 5.6(a)), the app disables the corresponding country name `Button`, uses an animation to *shake* the flag and displays **Incorrect!** in red at the bottom of the screen (Fig. 5.6(b)). Keep guessing until you get the correct answer for that flag.

Completing the Quiz

After you select the 10 correct country names, a popup `AlertDialog` displays over the app and shows your total number of guesses and the percentage of correct answers (Fig. 5.7). When you touch the dialog's **Reset Quiz** `Button`, a new quiz begins based on the current quiz options.

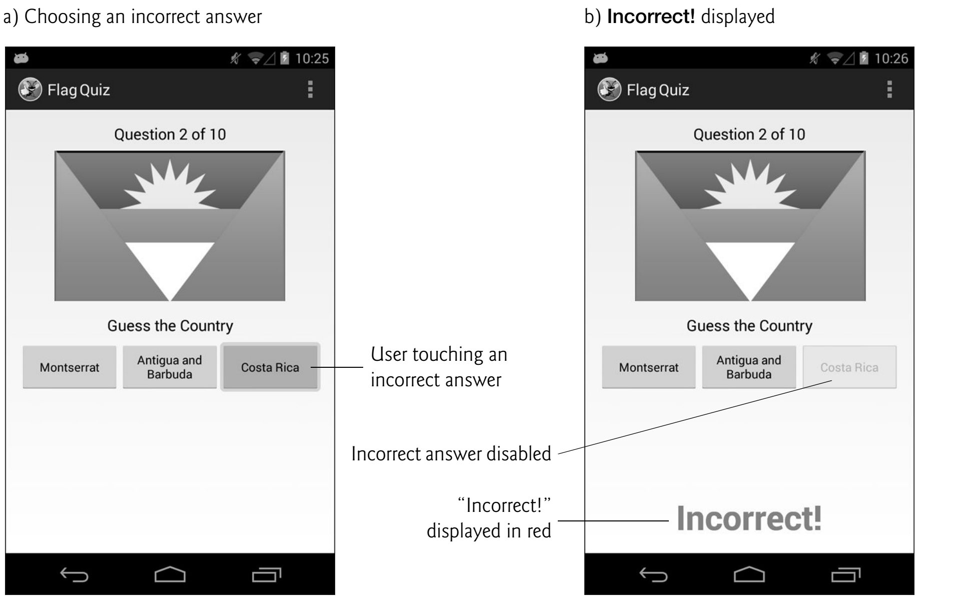

a) Choosing an incorrect answer
b) **Incorrect!** displayed
10:25
Flag Quiz
Question 2 of 10
Guess the Country
Montserrat
Antigua and Barbuda
Costa Rica
User touching an incorrect answer
10:26
Incorrect answer disabled
"Incorrect!" displayed in red
Incorrect!

Fig. 5.6 | Disabled incorrect answer in the **Flag Quiz** app.

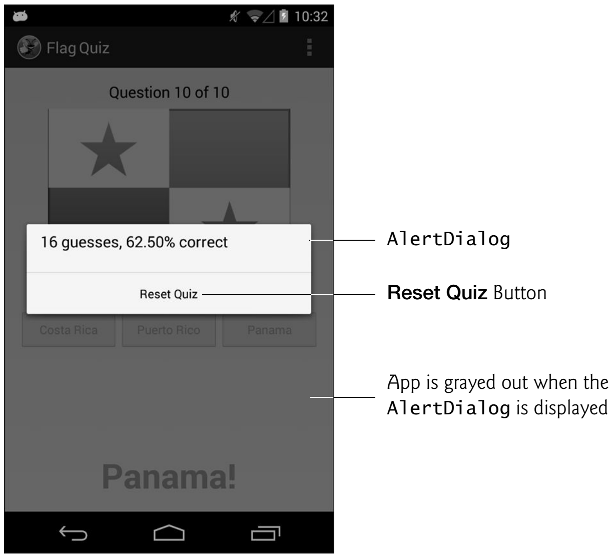


Fig. 5.7 | Results displayed after quiz completion.

5.3 Technologies Overview

This section introduces the features you'll use to build the **Flag Quiz** app.

5.3.1 Menus

When you create an app's project in the IDE, the `MainActivity` is configured to display an options menu (⋮) at the right side of the action bar. The menu contains a **Settings** menu item that's typically used to display an app's settings to the user. In later apps, you'll learn how to create additional menu items and how to decide which items should be displayed directly on the action bar vs. in the options menu.

The options menu is an object of class **`Menu`** (package `android.view`). To specify `Menu` options, you override `Activity`'s **`onCreateOptionsMenu`** method (Section 5.5.4) to add the options to the method's `Menu` argument. When the user selects a menu item, `Activity` method **`onOptionsItemSelected`** (Section 5.5.5) responds to the selection.

5.3.2 Fragments

A **fragment** typically represents a reusable portion of an `Activity`'s user interface, but may also represent reusable program logic. This app uses fragments to create and manage portions of the app's GUI. You can combine several fragments to create user interfaces that take advantage of tablet screen sizes. You also can easily interchange fragments to make your GUIs more dynamic—you'll learn about this in Chapter 8.

`Fragment` (package `android.app`) is the base class of all fragments. The **Flag Quiz** app defines the following direct and indirect `Fragment` subclasses:

- Class `QuizFragment` (Section 5.6)—a direct subclass of `Fragment`—displays the quiz's GUI and defines the quiz's logic. Like an `Activity`, each `Fragment` has its own layout that's typically defined as a layout resource, but can be created dynamically. In Section 5.4.8, you'll build `QuizFragment`'s GUI. You'll use the `QuizFragment` in `MainActivity`'s layouts—one for devices in portrait orientation and one for tablet devices in landscape orientation.
- Class `SettingsFragment` (Section 5.7) is a subclass of **`PreferenceFragment`** (package **`android.preference`**), which can automatically maintain an app's user preferences in a `SharedPreferences` file on the device. As you'll see, you can create an XML file that describes the user preferences and class `PreferenceFragment` can use that XML file to build an appropriate preferences GUI (Figs. 5.3–5.4).
- When you finish a quiz, the `QuizFragment` creates an anonymous inner class that extends **`DialogFragment`** (package `android.app`) and uses it to display an `AlertDialog` containing the quiz results (Section 5.6.9).

`Fragments` *must* be hosted by an `Activity`—they cannot execute independently. When this app runs in landscape orientation on a tablet, the `MainActivity` hosts all of the `Fragments`. In portrait orientation (on any device), the `SettingsActivity` (Section 5.8) hosts the `SettingsFragment` and the `MainActivity` hosts the others.

Though `Fragments` were introduced in Android 3.0, `Fragments` and other more recent Android features can be used in earlier versions via the Android Support Library. For more information, visit:

```
    http://developer.android.com/tools/support-library/index.html
```

5.3.3 Fragment Lifecycle Methods

Like an Activity, each Fragment has a *lifecycle* and provides methods that you can override to respond to lifecycle events. In this app, you'll override:

- **onCreate**—This method (which you'll override in class SettingsFragment) is called when a Fragment is created. The QuizFragment and SettingsFragment are created when their parent activities' layouts are inflated, and the DialogFragment that displays the quiz results is created when you complete a quiz.
- **onCreateView**—This method (which you'll override in class QuizFragment) is called after onCreate to build and return a View containing the Fragment's GUI. As you'll see, this method receives a **LayoutInflater**, which you'll use to programmatically inflate a Fragment's GUI from the components specified in a predefined XML layout.

We'll discuss other Fragment lifecycle methods as we encounter them throughout the book. For the complete lifecycle details, visit:

```
http://developer.android.com/guide/components/fragments.html
```

5.3.4 Managing Fragments

A parent Activity manages its Fragments with a **FragmentManager** (package android.app) that's returned by the Activity's **getFragmentManager** method. If the Activity needs to interact with a Fragment that's declared in the Activity's layout and has an **Id**, the Activity can call FragmentManager method **findFragmentById** to obtain a reference to the specified Fragment. As you'll see in Chapter 8, a FragmentManager can use **FragmentTransactions** (package android.app) to dynamically *add*, *remove* and *transition* between Fragments.

5.3.5 Preferences

In Section 5.2.2, you changed the app's settings to customize the quiz. A PreferenceFragment uses **Preference** objects (package android.preference) to manage these settings. This app uses Preference subclass **ListPreference** to manage the number of guess Buttons displayed for each flag and Preference subclass **MultiSelectListPreference** to manage the world regions to include in the quiz. A ListPreference's items are *mutually exclusive*, whereas any number of items can be selected in a MultiSelectListPreference. You'll use a **PreferenceManager** object (package android.preference) to access and interact with the app's preferences.

5.3.6 assets Folder

This app's flag images[1] are loaded into the app only when needed and are located in the app's **assets folder**. To add the images to the project, we dragged each region's folder from our file system onto the assets folder in the **Package Explorer**. The images are located in the images/FlagQuizImages folder with the book's examples.

Unlike an app's drawable folders, which require their image contents to be at the root level in each folder, the assets folder may contain files of any type and they can be organized in subfolders—we maintain the flag images for each region in a separate subfolder.

1. We obtained the images from www.free-country-flags.com.

Files in the `assets` subfolders are accessed via an **`AssetManager`** (package `android.content.res`), which can provide a list of all of the file names in a specified subfolder and can be used to access each asset.

5.3.7 Resource Folders

In Section 2.4.4, you learned about the `drawable`, `layout` and `values` subfolders of an app's `res` folder. In this app, you'll also use the `menu`, `anim` and `xml` resource folders. Figure 5.8 overviews these subfolders as well as the `animator`, `color` and `raw` subfolders.

Resource subfolder	Description
`anim`	Folder names that begin with `anim` contain XML files that define *tweened animations*, which can change an object's *transparency*, *size*, *position* and *rotation* over time. We'll define such an animation in Section 5.4.11 then play it in Section 5.6.9 to create a *shake effect* for visual feedback to the user.
`animator`	Folder names that begin with `animator` contain XML files that define *property animations*, which change the value of a property of an object over time. In Java, a property is typically implemented in a class as an instance variable with both *set* and *get* accessors.
`color`	Folder names that begin with `color` contain XML files that define a list of colors for various states, such as the states of a `Button` (*unpressed*, *pressed*, *enabled*, etc.).
`raw`	Folder names that begin with `raw` contain resource files (such as audio clips) that are read into an app as streams of bytes. We'll use such resources in Chapter 6 to play sounds.
`menu`	Folder names that begin with `menu` contain XML files that describe the contents of menus. When you create a project, the IDE automatically defines a menu with a **Settings** option.
`xml`	Folder names that begin with `xml` contain XML files that do not fit into the other resource categories. These are often raw XML data files used by the app. In Section 5.4.10, you'll create an XML file that represents the preferences displayed by this app's `SettingsFragment`.

Fig. 5.8 | Other subfolders within a project's `res` folder.

5.3.8 Supporting Different Screen Sizes and Resolutions

In Section 2.5.1 you learned that Android devices have various *screen sizes*, *resolutions* and *pixel densities* (dots per inch or DPI). You also learned that you typically provide images and other visual resources in multiple resolutions so Android can choose the best resource for a device's pixel density. Similarly, in Section 2.8, you learned how to provide string resources for different languages and regions. Android uses resource folders with *qualified names* to choose the appropriate images based on a device's pixel density and the correct language strings based on a device's locale and region settings. This mechanism also can be used to select resources from any of the resource folders discussed in Section 5.3.7.

For this app's `MainActivity`, you'll use size and orientation qualifiers to determine which layout to use—one for portrait orientation on phones and tablets and another for landscape orientation only on tablets. To do this, you'll define two `MainActivity` layouts:

- `activity_main.xml` in the project's `res/layout` folder is the default layout.
- `activity_main.xml` in the project's `res/layout-large-land` folder is used *only* on `large` devices (i.e., tablets) when the app is in landscape (`land`) orientation.

Qualified resource folder names have the format:

```
name-qualifiers
```

where *qualifiers* consists of one or more qualifiers separated by dashes (-). There are 18 types of qualifiers that you can add to resource folder names. We'll explain other qualifiers as we use them throughout the book. For a complete description of all the `res` subfolder qualifiers and the rules for the order in which they must be defined in a folder's name, visit:

```
http://developer.android.com/guide/topics/resources/
   providing-resources.html#AlternativeResources
```

5.3.9 Determining the Screen Size

In this app, we display the `Menu` only when the app is running on a phone-sized device or when it's running on a tablet in portrait orientation (Section 5.5.4). To determine this, we'll use Android's **`WindowManager`** (package `android.view`) to obtain a **`Display`** object that contains the display's current width and height. This changes with the device's orientation—in portrait orientation, the device's width is less than its height.

5.3.10 `Toasts` for Displaying Messages

A **`Toast`** (package `android.widget`) briefly displays a message, then disappears from the screen. `Toasts` are often used to display minor error messages or informational messages, such as that the quiz will be reset after the user changes the app's preferences. When the user changes the preferences, we display a `Toast` to indicate that the quiz will start over. We also display a `Toast` to indicate that at least one region must be selected if the user deselects all regions—in this case, the app sets North America as the default region for the quiz.

5.3.11 Using a `Handler` to Execute a `Runnable` in the Future

When the user makes a correct guess, the app displays the correct answer for two seconds before displaying the next flag. To do this, we use a **`Handler`** (package `android.os`). `Handler` method **`postDelayed`** receives as arguments a `Runnable` to execute and a delay in milliseconds. After the delay has passed, the `Handler`'s `Runnable` executes in the *same thread* that created the `Handler`. *Operations that interact with or modify the GUI must be performed in the GUI thread, because GUI components are not thread safe.*

5.3.12 Applying an Animation to a `View`

When the user makes an incorrect choice, the app shakes the flag by applying an **`Animation`** (package `android.view.animation`) to the `ImageView`. We use **`AnimationUtils`** `static` method **`loadAnimation`** to load the animation from an XML file that specifies the animation's options. We also specify the number of times the animation should repeat with `Animation` method **`setRepeatCount`** and perform the animation by calling `View` method **`startAnimation`** (with the `Animation` as an argument) on the `ImageView`.

5.3.13 Logging Exception Messages

When exceptions occur, you can *log* them for debugging purposes with Android's built-in logging mechanism. Android provides class **`Log`** (package `android.util`) with several `static` methods that represent messages of varying detail. Logged messages can be viewed in the **LogCat** tab at the bottom of the IDE as well as with the **Android `logcat` tool**. For more details on logging messages, visit

```
http://developer.android.com/reference/android/util/Log.html
```

5.3.14 Using an Explicit `Intent` to Launch Another `Activity` in the Same App

When this app runs in portrait orientation, the app's preferences are displayed in the `SettingsActivity` (Section 5.8). In Chapter 4, we showed how to use an *implicit* `Intent` to display a URL in the device's web browser. Section 5.5.5 shows how to use an **explicit `Intent`** to launch a specific `Activity` in the same app.

5.3.15 Java Data Structures

This app uses various data structures from the `java.util` package. The app dynamically loads the image file names for the enabled regions and stores them in an `ArrayList<String>`. We use `Collections` method `shuffle` to randomize the order of the image file names for each new game. We use a second `ArrayList<String>` to hold the image file names of the countries in the current quiz. We also use a **`Set<String>`** to store the world regions included in a quiz. We refer to the `ArrayList<String>` object with a variable of interface type `List<String>`—this is a good Java programming practice that enables you to change data structures easily without affecting the rest of your app's code.

5.4 Building the GUI and Resource Files

In this section, you'll create the project and configure the `String`, array, color, dimension, layout and animation resources used by the **Flag Quiz** app.

5.4.1 Creating the Project

Before you create the new project, delete the `FlagQuiz` project that you test-drove in Section 5.2 by right clicking it and selecting **Delete**. In the dialog that appears, ensure that **Delete project contents on disk** is *not* selected, then click **OK**.

Creating a New Blank App Project

Next, create a new **Android Application Project**. Specify the following values in the **New Android Project** dialog's first **New Android Application** step, then press **Next >**:

- **Application name:** `Flag Quiz`
- **Project name:** `FlagQuiz`
- **Package name:** `com.deitel.flagquiz`

- **Minimum Required SDK:** `API18: Android 4.3`
- **Target SDK:** `API19: Android 4.4`
- **Compile With:** `API19: Android 4.4`
- **Theme:** `Holo Light with Dark Action Bar`

In the **New Android Project** dialog's second **New Android Application** step, leave the default settings, and press **Next >**. In the **Configure Launcher Icon** step, click the **Browse...** button, and select an app icon image (provided in the `images` folder with the book's examples), press **Open** then **Next >**. In the **Create Activity** step, select **Blank Activity**, then press **Next >**. In the **Blank Activity** step, leave the default settings and click **Finish** to create the project. Open `activity_main.xml` in the **Graphical Layout** editor and select **Nexus 4** from the screen-type drop-down list. Once again, we'll use this device as the basis for our design.

5.4.2 `strings.xml` and Formatted `String` Resources

You created `String` resources in earlier chapters, so we show only a table (Fig. 5.9) of the `String` resource names and corresponding values here. Double click `strings.xml` in the `res/values` folder to display the resource editor for creating these `String` resources.

Resource name	Value
`settings_activity`	`Flag Quiz Settings`
`number_of_choices`	`Number of Choices`
`number_of_choices_description`	`Display 3, 6 or 9 guess buttons`
`world_regions`	`Regions`
`world_regions_description`	`Regions to include in the quiz`
`guess_country`	`Guess the Country`
`results`	`%1$d guesses, %2$.02f%% correct`
`incorrect_answer`	`Incorrect!`
`default_region_message`	`Setting North America as the default region. One region must be selected.`
`restarting_quiz`	`Quiz will restart with your new settings`
`ok`	`OK`
`question`	`Question %1$d of %2$d`
`reset_quiz`	`Reset Quiz`
`image_description`	`Image of the current flag in the quiz`
`default_region`	`North_America`

Fig. 5.9 | `String` resources used in the **Flag Quiz** app.

Format `Strings` *as* `String` *Resources*

The `results` and `question` resources are format `Strings` that are used with `String` method `format`. When a `String` resource contains multiple format specifiers you must number the format specifiers for localization purposes. In the `results` resource, the notation `1$` in `%1$d` indicates that `String` method `format`'s *first* argument after the format `String` should replace the format specifier %1$d. Similarly, 2$ in %2$.02f indicates that the *second* argument after the format String should replace the format specifier %2$.02f. The d in the first format specifier formats an integer and the f in the second one formats a floating-point number. In localized versions of strings.xml, the format specifiers %1$d and %2$.02f can be reordered as necessary to properly translate the String resource. The *first* argument after the format String will replace %1$d—regardless of where it appears in the format String—and the *second* argument will replace %2$.02f *regardless* of where they appear in the format String.

5.4.3 arrays.xml

In Section 4.5.8, you created a String array resource in the app's strings.xml file. Technically, all of your app's resources in the res/values folder can be defined in the *same* file. However, to make it easier to manage different types of resources, separate files are typically used for each. For example, array resources are normally defined in arrays.xml, colors in colors.xml, Strings in strings.xml and numeric values in values.xml. This app uses three String array resources that are defined in arrays.xml:

- regions_list specifies the names of the world regions with their words separated by underscores—these values are used to load image file names from the appropriate folders and as the selected values for the world regions the user selects in the SettingsFragment.
- regions_list_for_settings specifies the names of the world regions with their words separated by spaces—these values are used in the SettingsFragment to display the region names to the user.
- guesses_list specifies the Strings 3, 6 and 9—these values are used in the SettingsFragment to display the options for the number of guess Buttons to display.

Figure 5.10 shows the names and element values for these three array resources.

Array resource name	Values
regions_list	Africa, Asia, Europe, North_America, Oceania, South_America
regions_list_for_settings	Africa, Asia, Europe, North America, Oceania, South America
guesses_list	3, 6, 9

Fig. 5.10 | String array resources defined in arrays.xml.

To create the file and configure the array resources, perform the following steps:

1. In the project's res folder, right click the values folder, then select **New > Android XML File** to display the **New Android XML File** dialog. Because you right clicked the values folder, the dialog is preconfigured to add a **Values** resource file in the values folder.
2. Specify arrays.xml in the **File** field and click **Finish** to create the file.

3. If the IDE opens the new file in XML view, click the **Resources** tab at the bottom of the window to view the resource editor.
4. To create a `String` array resource, click **Add...**, select **String Array** and click **OK**.
5. In the **Name** field, enter `regions_list`, then save the file.
6. Select the new `String` array resource, then use the **Add** button to add items for each of the values shown for the array in Fig. 5.10.
7. Repeat Steps 4–6 for the `regions_list_for_settings` and `guesses_list` arrays. When you click **Add...** to create the additional **String Array** resources, you'll need to first select the radio button **Create a new element at the top level in Resources**.

5.4.4 `colors.xml`

This app displays correct answers in green and incorrect answers in red. As with any other resource, color resources should be defined in XML so that you can easily change colors without modifying your app's Java source code. Typically, colors are defined in a file name `colors.xml`, which you must create. As you learned in Section 3.4.5, colors are defined using the RGB or ARGB color schemes.

To create the file and configure the two color resources, perform the following steps:

1. In the project's `res` folder, right click the `values` folder, then select **New > Android XML File** to display the **New Android XML File** dialog.
2. Specify `colors.xml` in the **File** field and click **Finish** to create the file.
3. If the IDE opens the new file in XML view, click the **Resources** tab at the bottom of the window to view the resource editor.
4. To create a color resource, click **Add...**, select **Color** and click **OK**.
5. In the **Name** and **Value** fields that appear, enter `correct_answer` and `#00CC00`, respectively, then save the file.
6. Repeat Steps 4 and 5, but enter `incorrect_answer` and `#FF0000`.

5.4.5 `dimens.xml`

You created dimension resources in earlier chapters, so we show only a table (Fig. 5.11) of the dimension resource names and values here. Open `dimens.xml` in the `res/values` folder to display the resource editor for creating these resources. The `spacing` resource is used in the layouts as the spacing between various GUI components, and the `answer_size` resource specifies the font size for the `answerTextView`. Recall from Section 2.5.3 that font sizes should be specified in scale-independent pixels (`sp`) so that fonts in your app can also be scaled by the user's preferred font size (as specified in the device's settings).

Resource name	Value
`spacing`	`8dp`
`answer_size`	`40sp`

Fig. 5.11 | Dimension resources used in the **Flag Quiz** app.

5.4.6 `activity_settings.xml` Layout

In this section, you'll create the layout for the `SettingsActivity` (Section 5.8) that will display the `SettingsFragment` (Section 5.7). The `SettingsActivity`'s layout will consist of only a `LinearLayout` containing the GUI for the `SettingsFragment`. As you'll see, when you add a `Fragment` to a layout, the IDE can create the `Fragment`'s class for you. To create this layout, perform the following steps:

1. In the project's `res` folder, right click `layout` and select **New > Android XML File** to display the **New Android XML File** dialog. Because you right clicked the `layout` folder, the dialog is preconfigured to add a **Layout** resource file.
2. In the **File** field, enter `activity_settings.xml`.
3. In the **Root Element** section, select **LinearLayout** and click **Finish** to create the file.
4. From the **Palette**'s **Layouts** section, drag a **Fragment** onto the design area or onto the `LinearLayout` node in the **Outline** window.
5. The preceding step displays the **Choose Fragment Class** dialog. If you defined the `Fragment` class before its layout, you'd be able to select the class here. Click **Create New...** to display the **New Java Class** dialog.
6. Enter `SettingsFragment` in the dialog's **Name** field, change the **Superclass** field's value to `android.preference.PreferenceFragment` and click **Finish** to create the class. The IDE opens the Java file for the class, which you can close for now.
7. Save `activity_settings.xml`.

5.4.7 `activity_main.xml` Layout for Phone and Tablet Portrait Orientation

In this section, you'll create the layout for the `MainActivity` (Section 5.5) that will be used in portrait orientation on all devices. You'll define the landscape orientation layout for tablets in Section 5.4.9. This layout will display only the `QuizFragment` (Section 5.6):

1. In the project's `res/layout` folder, open `activity_main.xml`, then follow the steps in Section 2.5.2 to switch from a `FrameLayout` to a `RelativeLayout`.
2. From the **Palette**'s **Layouts** section, drag a **Fragment** onto the `RelativeLayout` node in the **Outline** window.
3. In the **Choose Fragment Class** dialog, click **Create New...** to display the **New Java Class** dialog.
4. In the dialog's **Name** field, enter `QuizFragment`, then click **Finish** to create the class. The IDE opens the Java file for the class, which you can close for now.
5. In `activity_main.xml`, select the `QuizFragment` in the **Outline** window, then set its **Id** to `@+id/quizFragment` and, in the **Layout Parameters** properties, set **Width** and **Height** to `match_parent`.
6. Save `activity_main.xml`.

5.4.8 `fragment_quiz.xml` Layout

You'll typically define a layout for each of your `Fragments`. For each `Fragment` layout, you'll add a layout XML file to your app's `res/layout` folder(s) and specify which `Frag-`

ment class the layout is associated with. Note that you do not need to define a layout for this app's SettingsFragment because its GUI is auto-generated by the inherited capabilities of class PreferenceFragment.

This section presents the QuizFragment's layout (fragment_quiz.xml). You'll define its layout file once in the app's res/layout folder, because we use the same layout for the QuizFragment on all devices and device orientations. Figure 5.12 shows the QuizFragment's GUI component names.

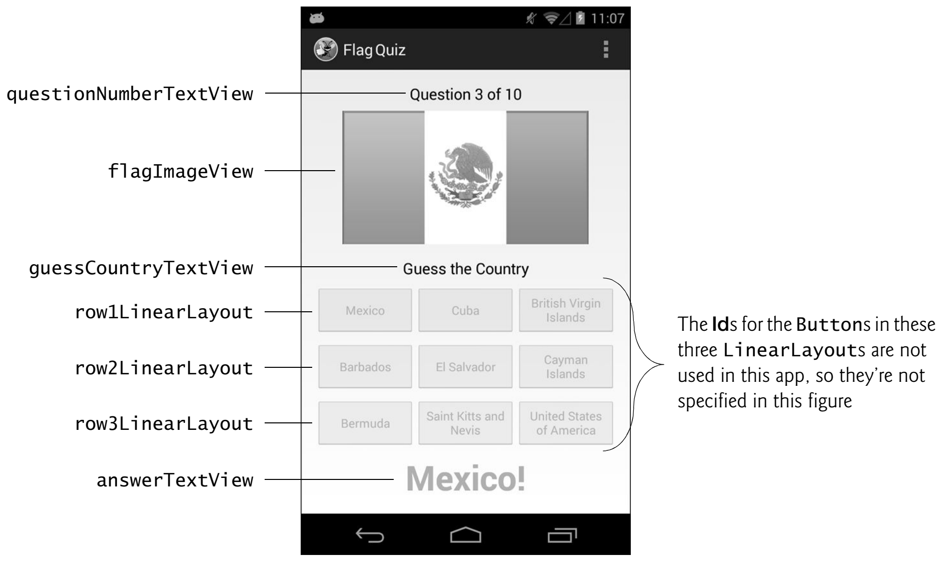
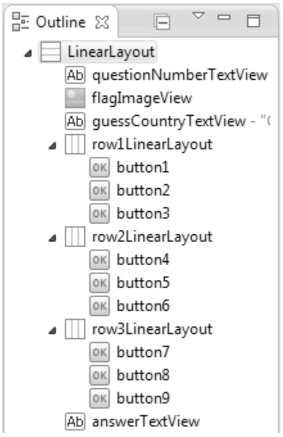


Fig. 5.12 | **Flag Quiz** GUI's components labeled with their **Id** property values.

Creating ***fragment_quiz.xml***

To create fragment_quiz.xml, perform the following steps:

1. In the project's res folder, right click the layout folder, then select **New > Android XML File** to display the **New Android XML File** dialog.
2. In the **File** field, enter fragment_quiz.xml.
3. In the **Root Element** section, select **LinearLayout (Vertical)** and click **Finish** to create the layout file.
4. Use the **Graphical Layout** editor and the **Outline** window to form the layout structure shown in Fig. 5.13. As you create the GUI components, set their **Id** properties. For the questionNumberTextView and guessCountryTextView, we used **Medium Text** components from the **Palette**'s **Form Widgets** section. For the Buttons, we used **Small Button** components, which use a smaller font size so that they can fit more text.
5. Once you've completed Step 4, configure the GUI component properties with the values shown in Fig. 5.14. Setting flagImageView's **Height** to 0dp and **Weight**



```
LinearLayout
    questionNumberTextView
    flagImageView
    guessCountryTextView - "(
    row1LinearLayout
        button1
        button2
        button3
    row2LinearLayout
        button4
        button5
        button6
    row3LinearLayout
        button7
        button8
        button9
    answerTextView
```

Fig. 5.13 | **Outline** window for `fragment_quiz.xml`.

to `1` enables this component to resize vertically to occupy any remaining space not used by the other GUI components. Similarly, setting each `Button`'s **Width** to `0dp` and **Weight** to `1` enables the `Buttons` in a given `LinearLayout` to divide the horizontal space equally. The `flagImageView`'s **Scale Type** value `fitCenter` scales the image to fill either the `ImageView`'s width or height while maintaining the original image's aspect ratio. Setting the `ImageView`'s **Adjust View Bounds** property to `true` indicates that the `ImageView` maintains the aspect ratio of its `Drawable`.

GUI component	Property	Value
`questionNumberTextView`	*Layout Parameters*	
	Width	`wrap_content`
	Height	`wrap_content`
	Gravity	`center_horizontal`
	Other Properties	
	Text	`@string/question`
`flagImageView`	*Layout Parameters*	
	Width	`wrap_content`
	Height	`0dp`
	Gravity	`center`
	Weight	`1`
	Margins	
	Left/Right	`@dimen/activity_horizontal_margin`
	Top/Bottom	`@dimen/activity_vertical_margin`
	Other Properties	
	Adjust View Bounds	`true`
	Content Desctiption	`@string/image_description`
	Scale Type	`fitCenter`

Fig. 5.14 | Property values for the GUI components in `fragment_quiz.xml`. (Part 1 of 2.)

GUI component	Property	Value
guessCountryTextView	*Layout Parameters*	
	Width	wrap_content
	Height	wrap_content
	Gravity	center_horizontal
	Other Properties	
	Text	@string/guess_country
LinearLayouts	*Layout Parameters*	
	Width	match_parent
	Height	wrap_content
	Margins	
	Bottom	@dimen/spacing
Buttons	*Layout Parameters*	
	Width	0dp
	Height	fill_parent
	Weight	1
answerTextView	*Layout Parameters*	
	Width	wrap_content
	Height	wrap_content
	Gravity	center|bottom
	Other Properties	
	Gravity	center_horizontal
	Text Size	@dimen/answer_size
	Text Style	bold

Fig. 5.14 | Property values for the GUI components in `fragment_quiz.xml`. (Part 2 of 2.)

5.4.9 `activity_main.xml` Layout for Tablet Landscape Orientation

In Section 5.4.7, you defined `MainActivity`'s portrait-orientation layout, which contained only the `QuizFragment`. You'll now define `MainActivity`'s landscape-orientation layout for tablets, which will contain both the `SettingsFragment` and the `QuizFragment`. To create the layout, perform the following steps:

1. Right click the project's `res` folder, then select **New > Folder**. In the **Folder name** field enter `layout-large-land` and click **Finish**. The qualifiers `large` and `land` ensure that any layouts defined in this folder will be used only on large devices on which the app is running in landscape orientation.
2. Right click the `layout-large-land` folder, then select **New > Android XML File** to display the **New Android XML File** dialog, then enter `activity_main.xml` in **File** field. In the **Root Element** section, select **LinearLayout (Horizontal)** and click **Finish** to create the layout file.
3. Select the `LinearLayout` and set its **Base Aligned** property to `false`.
4. From the **Layouts** section of the **Graphical Layout** editor, drag a **Fragment** onto the `LinearLayout` node in the **Outline** window. In the **Choose Fragment Class** dialog, select `SettingsFragment` and click **OK**.

5. Repeat Step 5, but select `QuizFragment` and click **OK**.
6. Select the `SettingsFragment` node in the **Outline** window. In the **Layout Parameters** section set **Width** to `0dp`, **Height** to `match_parent` and **Weight** to `1`.
7. Select the `QuizFragment` node in the **Outline** window. In the **Layout Parameters** section set **Width** to `0dp`, **Height** to `match_parent` and **Weight** to `2`.

Because the `QuizFragment`'s **Weight** is `2` and the `SettingsFragment`'s is `1`, the `QuizFragment` will occupy two-thirds of the layout's horizontal space.

5.4.10 `preferences.xml` for Specifying the App's Settings

In this section, you'll create the `preferences.xml` file that the `SettingsFragment` uses to display the app's preferences. To create the file:

1. Right click the project's `res` folder, then select **New > Folder**, in the **Folder name** field enter `xml` and click **Finish**.
2. Right click the `xml` folder, then select **New > Android XML File** to display the **New Android XML File** dialog.
3. In the **File** text field, enter the name `preferences.xml`.
4. Ensure that the **Resource Type** is set to **Preference** and the **Root Element** is **PreferenceScreen**, which represents a screen in which preferences are displayed.
5. Click **Finish** to create the file. If the IDE displays the raw XML, click the **Structure** tab at the bottom of the window to configure the preferences.
6. At the left side of the window, select `PreferenceScreen`, then click **Add...**.
7. In the dialog that appears, select **ListPreference**, then click **OK**. This preference will display a list of mutually exclusive options.
8. At the left side of the window, select **PreferenceScreen**, then click **Add...**.
9. In the dialog that appears, select **MultiSelectListPreference**, then click **OK**. This preference will display a list of options in which multiple items can be selected. All of the selected items are saved as the value of such a preference.
10. Select the **ListPreference**, then configure the properties in Fig. 5.15.
11. Select the **MultiSelectListPreference**, then configure the properties in Fig. 5.16.
12. Save and close `preferences.xml`.

Property	Value	Description
Entries	`@array/guesses_list`	An array of `Strings` that will be displayed in the list of options.
Entry values	`@array/guesses_list`	An array of the values associated with the options in the **Entries** property. The selected entry's value will be stored in the app's `SharedPreferences`.

Fig. 5.15 | `ListPreference` property values. (Part 1 of 2.)

Property	Value	Description
Key	`pref_numberOfChoices`	The name of the preference stored in the app's `SharedPreferences`.
Title	`@string/number_of_choices`	The title of the preference displayed in the GUI.
Summary	`@string/number_of_choices_description`	A summary description of the preference that's displayed below its title.
Persistent	`true`	Whether the preference should persist after the app terminates—if `true`, class `PreferenceFragment` immediately persists the preference value each time it changes.
Default value	`3`	The item in the **Entries** property that's selected by default.

Fig. 5.15 | `ListPreference` property values. (Part 2 of 2.)

Property	Value	Description
Entries	`@array/regions_list_for_settings`	An array of `Strings` that will be displayed in the list of options.
Entry values	`@array/regions_list`	An array of the values associated with the options in the **Entries** property. The selected entries' values will *all* be stored in the app's `SharedPreferences`.
Key	`pref_regionsToInclude`	The name of the preference stored in the app's `SharedPreferences`.
Title	`@string/world_regions`	The title of the preference displayed in the GUI.
Summary	`@string/world_regions_description`	A summary description of the preference that's displayed below its title.
Persistent	`true`	Whether the preference should persist after the app terminates.
Default value	`@array/regions_list`	An array of the default values for this preference—in this case, all of the regions will be selected by default.

Fig. 5.16 | `MultiSelectListPreference` property values.

5.4.11 Creating the Flag Shake Animation

In this section, you'll create the animation that shakes the flag when the user guesses incorrectly. We'll show how this animation is used by the app in Section 5.6.9. To create the animation:

1. Right click the project's `res` folder, then select **New > Folder**, in the **Folder name** field enter `anim` and click **Finish**.
2. Right click the `anim` folder, then select **New > Android XML File** to display the **New Android XML File** dialog.
3. In the **File** text field, enter the name `incorrect_shake.xml`.
4. Ensure that the **Resource Type** is **Tween Animation** and the **Root Element** is **set**.
5. Click **Finish** to create the file. The file opens immediately in **XML** view.
6. Unfortunately, the IDE does not provide an editor for animations, so you must modify the XML contents of the file as shown in Fig. 5.17.

```
 1  <?xml version="1.0" encoding="utf-8"?>
 2
 3  <set xmlns:android="http://schemas.android.com/apk/res/android"
 4     android:interpolator="@android:anim/decelerate_interpolator">
 5
 6     <translate android:fromXDelta="0" android:toXDelta="-5%p"
 7        android:duration="100"/>
 8
 9     <translate android:fromXDelta="-5%p" android:toXDelta="5%p"
10        android:duration="100" android:startOffset="100"/>
11
12     <translate android:fromXDelta="5%p" android:toXDelta="-5%p"
13        android:duration="100" android:startOffset="200"/>
14  </set>
```

Fig. 5.17 | Shake animation (`incorrect_shake.xml`) that's applied to the flag when the user guesses incorrectly.

In this example, we use **`View` animations** to create a *shake effect* that consists of three animations in an **animation `set`** (lines 3–14)—a collection of animations that make up a larger animation. Animation `set`s may contain any combination of **tweened animations—`alpha`** (transparency), **`scale`** (resize), **`translate`** (move) and **`rotate`**. Our shake animation consists of a series of three `translate` animations. A `translate` animation moves a `View` within its parent. Android also supports *property animations* in which you can animate any property of any object.

The first `translate` animation (lines 6–7) moves a `View` from a starting location to an ending position over a specified period of time. The **`android:fromXDelta` attribute** is the `View`'s offset when the animation starts and the **`android:toXDelta` attribute** is the `View`'s offset when the animation ends. These attributes can have

- absolute values (in pixels)
- a percentage of the animated `View`'s size
- a percentage of the animated `View`'s *parent's* size.

For the `android:fromXDelta` attribute, we specified an absolute value of 0. For the `android:toXDelta` attribute, we specified the value `-5%p`, which indicates that the `View` should move to the *left* (due to the minus sign) by 5% of the parent's width (indicated by the `p`). If we wanted to move by 5% of the `View`'s width, we would leave out the `p`. The **`android:duration` attribute** specifies how long the animation lasts in milliseconds. So the animation in lines 6–7 will move the `View` to the left by 5% of its parent's width in 100 milliseconds.

The second animation (lines 9–10) continues from where the first finished, moving the `View` from the `-5%p` offset to a `%5p` offset in 100 milliseconds. By default, animations in an animation `set` are applied simultaneously (i.e., in parallel), but you can use the **`android:startOffset` attribute** to specify the number of milliseconds into the future at which an animation should begin. This can be used to sequence the animations in a `set`. In this case, the second animation starts 100 milliseconds after the first. The third animation (lines 12–13) is the same as the second, but in the reverse direction, and it starts 200 milliseconds after the first animation.

5.5 `MainActivity` Class

Class `MainActivity` (Figs. 5.18–Fig. 5.23) hosts the app's `QuizFragment` when the app is running in portrait orientation, and hosts both the `SettingsFragment` and `QuizFragment` when the app is running on a tablet in landscape orientation.

5.5.1 package Statement, `import` Statements and Fields

Figure 5.18 shows the `MainActivity` `package` statement, `import` statements and fields. Lines 6–21 `import` the various Java and Android classes and interfaces that the app uses. We've highlighted the new `import` statements, and we discuss the corresponding classes and interfaces in Section 5.3 and as they're encountered in Sections 5.5.2—5.5.6.

```
 1  // MainActivity.java
 2  // Hosts the QuizFragment on a phone and both the
 3  // QuizFragment and SettingsFragment on a tablet
 4  package com.deitel.flagquiz;
 5
 6  import java.util.Set;
 7
 8  import android.app.Activity;
 9  import android.content.Intent;
10  import android.content.SharedPreferences;
11  import android.content.SharedPreferences.OnSharedPreferenceChangeListener;
12  import android.content.pm.ActivityInfo;
13  import android.content.res.Configuration;
14  import android.graphics.Point;
15  import android.os.Bundle;
16  import android.preference.PreferenceManager;
17  import android.view.Display;
18  import android.view.Menu;
19  import android.view.MenuItem;
20  import android.view.WindowManager;
21  import android.widget.Toast;
```

Fig. 5.18 | `MainActivity` `package` statement, `import` statements and fields. (Part 1 of 2.)

```
22
23 public class MainActivity extends Activity
24 {
25    // keys for reading data from SharedPreferences
26    public static final String CHOICES = "pref_numberOfChoices";
27    public static final String REGIONS = "pref_regionsToInclude";
28
29    private boolean phoneDevice = true; // used to force portrait mode
30    private boolean preferencesChanged = true; // did preferences change?
31
```

Fig. 5.18 | `MainActivity` `package` statement, `import` statements and fields. (Part 2 of 2.)

Lines 26–27 define constants for the preference keys you created in Section 5.4.10. You'll use these to access the corresponding preference values. The `boolean` variable `phoneDevice` (line 29) specifies whether the app is running on a phone—if so, the app will allow only portrait orientation. The `boolean` variable `preferencesChanged` (line 30) specifies whether the app's preferences have changed—if so, the `MainActivity`'s `onStart` lifecycle method (Section 5.5.3) will call the `QuizFragment`'s methods `updateGuessRows` (Section 5.6.4) and `updateRegions` (Section 5.6.5) to reconfigure the quiz based on the user's new settings. We set this `boolean` to `true` initially so that when the app first executes the quiz is configured using the default preferences.

5.5.2 Overridden `Activity` Method onCreate

Overridden `Activity` method `onCreate` (Fig. 5.19) calls `setContentView` (line 36) to set `MainActivity`'s GUI. Android chooses the `activity_main.xml` file from the `res/layout` folder if the app is running in portrait orientation or `res/layout-large-land` if the app is running on a tablet in landscape orientation.

```
32    @Override
33    protected void onCreate(Bundle savedInstanceState)
34    {
35       super.onCreate(savedInstanceState);
36       setContentView(R.layout.activity_main);
37
38       // set default values in the app's SharedPreferences
39       PreferenceManager.setDefaultValues(this, R.xml.preferences, false);
40
41       // register listener for SharedPreferences changes
42       PreferenceManager.getDefaultSharedPreferences(this).
43          registerOnSharedPreferenceChangeListener(
44             preferenceChangeListener);
45
46       // determine screen size
47       int screenSize = getResources().getConfiguration().screenLayout &
48          Configuration.SCREENLAYOUT_SIZE_MASK;
49
```

Fig. 5.19 | `MainActivity` overridden `Activity` method `onCreate`. (Part 1 of 2.)

```
50        // if device is a tablet, set phoneDevice to false
51        if (screenSize == Configuration.SCREENLAYOUT_SIZE_LARGE ||
52           screenSize == Configuration.SCREENLAYOUT_SIZE_XLARGE )
53           phoneDevice = false; // not a phone-sized device
54
55        // if running on phone-sized device, allow only portrait orientation
56        if (phoneDevice)
57           setRequestedOrientation(
58              ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
59     } // end method onCreate
60
```

Fig. 5.19 | `MainActivity` overridden `Activity` method `onCreate`. (Part 2 of 2.)

Setting the Default Preference Values and Registering a Change Listener

When you install and launch the app for the first time, line 39 sets the app's *default preferences* by calling `PreferenceManager` method **`setDefaultValues`**—this creates and initializes the app's `SharedPreferences` file using the default values that you specified in `preferences.xml`. The method requires three arguments:

- the preferences' `Context`—the `Activity` (`this`) for which you are setting the default preferences
- the resource ID for the preferences XML file (`R.xml.preferences`) that you created in Section 5.4.10
- a `boolean` indicating whether the default values should be reset each time method `setDefaultValues` is called—`false` indicates that the default preference values should be set only the first time this method is called.

Each time the user changes the app's preferences, `MainActivity` should call `QuizFragment`'s methods `updateGuessRows` or `updateRegions` (based on which preference changed) to reconfigure the quiz. `MainActivity` registers an `OnSharedPreferenceChangedListener` (lines 42–44) so that it will be notified each time a preference changes. `PreferenceManager` method **`getDefaultSharedPreferences`** returns a reference to the `SharedPreferences` object representing the app's preferences, and `SharedPreferences` method **`registerOnSharedPreferenceChangeListener`** registers the listener, which is defined in Section 5.5.6.

Configuring a Phone Device for Portrait Orientation

Lines 47–53 determine whether the app is running on a tablet or a phone. Inherited method **`getResources`** returns the app's **`Resources`** object (package `android.content.res`) that can be used to access an app's resources and determine information about the app's environment. `Resources` method **`getConfiguration`** returns a **`Configuration`** object (package `android.content.res`) that contains `public` instance variable `screenLayout`, which you can use to determine the device's screen-size category. To do so, first you combine the value of `screenLayout` with `Configuration.SCREENLAYOUT_SIZE_MASK` using the bitwise AND (&) operator. Then, you compare the result to the `Configuration` constants `SCREENLAYOUT_SIZE_LARGE` and `SCREENLAYOUT_SIZE_XLARGE` (lines 51–52). If either is a match, the app is running on a tablet-sized device. Finally, if the device is a phone, lines 57–58 call inherited `Activity` method **`setRequestedOrientation`** to force the app to display `MainActivity` in only portrait orientation.

5.5.3 Overridden Activity Method onStart

Overridden Activity lifecycle method **onStart** (Fig. 5.20) is called in two scenarios:

- When the app first executes, onStart is called after onCreate. We use onStart in this case to ensure that the quiz is configured correctly based on the app's default preferences when the app is installed and executes for the first time or based on the user's updated preferences when the app is launched subsequently.
- When the app is running in portrait orientation and the user opens the SettingsActivity, the MainActivity is *paused* while the SettingsActivity is displayed. When the user returns to the MainActivity, onStart is called again. We use onStart in this case to ensure that the quiz is reconfigured properly if the user made any preference changes.

In both cases, if preferencesChanged is true, onStart calls QuizFragment's updateGuessRows (Section 5.6.4) and updateRegions (Section 5.6.5) methods to reconfigure the quiz. To get a reference to the QuizFragment so we can call its methods, lines 71–72 use inherited Activity method getFragmentManager to get the FragmentManager, then call its findFragmentById method. Next, lines 73–76 call QuizFragment's updateGuessRows and updateRegions methods, passing the app's SharedPreferences object as an argument so those methods can load the current preferences. Line 77 resets the quiz.

```
61     // called after onCreate completes execution
62     @Override
63     protected void onStart()
64     {
65        super.onStart();
66
67        if (preferencesChanged)
68        {
69           // now that the default preferences have been set,
70           // initialize QuizFragment and start the quiz
71           QuizFragment quizFragment = (QuizFragment)
72              getFragmentManager().findFragmentById(R.id.quizFragment);
73           quizFragment.updateGuessRows(
74              PreferenceManager.getDefaultSharedPreferences(this));
75           quizFragment.updateRegions(
76              PreferenceManager.getDefaultSharedPreferences(this));
77           quizFragment.resetQuiz();
78           preferencesChanged = false;
79        }
80     } // end method onStart
81
```

Fig. 5.20 | MainActivity overridden Activity method onStart.

5.5.4 Overridden Activity Method onCreateOptionsMenu

We override Activity method onCreateOptionsMenu (Fig. 5.21) to initialize Activity's standard options menu. The system passes in the Menu object where the options will appear. In this app, we want to show the menu only when the app is running in portrait orientation. Lines 87–88 use the WindowManager to get a Display object that contains the screen's current width and height, which changes based on the device's orientation. If the width is less than the height, then the device is in portrait orientation. Line 89 creates a `Point` object to store the current width and height, then line 90 calls `Display` method `getRealSize`, which stores the screen's width and height in the `Point`'s `public` instance variables `x` and `y`, respectively. If the width is less than the height (line 93), line 95 creates the menu from `menu.xml`—the default menu resource that the IDE configured when you created the project. Inherited `Activity` method **`getMenuInflater`** returns a **`MenuInflater`** on which we call **`inflate`** with two arguments—the resource ID of the menu resource that populates the menu and the `Menu` object in which the menu items will be placed. Returning true from `onCreateOptionsMenu` indicates that the menu should be displayed.

```
 82     // show menu if app is running on a phone or a portrait-oriented tablet
 83     @Override
 84     public boolean onCreateOptionsMenu(Menu menu)
 85     {
 86        // get the default Display object representing the screen
 87        Display display = ((WindowManager)
 88           getSystemService(WINDOW_SERVICE)).getDefaultDisplay();
 89        Point screenSize = new Point(); // used to store screen size
 90        display.getRealSize(screenSize); // store size in screenSize
 91
 92        // display the app's menu only in portrait orientation
 93        if (screenSize.x < screenSize.y) // x is width, y is height
 94        {
 95           getMenuInflater().inflate(R.menu.main, menu); // inflate the menu
 96           return true;
 97        }
 98        else
 99           return false;
100     } // end method onCreateOptionsMenu
101
```

Fig. 5.21 | `MainActivity` overridden `Activity` method `onCreateOptionsMenu`.

5.5.5 Overridden `Activity` Method `onOptionsItemSelected`

Method `onOptionsItemSelected` (Fig. 5.22) is called when a menu item is selected. In this app, the default menu provided by the IDE when you created the project contains only the **Settings** menu item, so if this method is called, the user selected **Settings**. Line 106 creates an explicit `Intent` for launching the `SettingsActivity`. The `Intent` constructor used here receives the `Context` from which the `Activity` will be launched and the class representing the `Activity` to launch (`SettingsActivity.class`). We then pass this `Intent` to the inherited `Activity` method `startActivity` to launch the `Activity`.

```
102     // displays SettingsActivity when running on a phone
103     @Override
104     public boolean onOptionsItemSelected(MenuItem item)
105     {
```

Fig. 5.22 | `MainActivity` overridden `Activity` method `onOptionsItemSelected`. (Part 1 of 2.)

```
106          Intent preferencesIntent = new Intent(this, SettingsActivity.class);
107          startActivity(preferencesIntent);
108          return super.onOptionsItemSelected(item);
109       }
110
```

Fig. 5.22 | `MainActivity` overridden `Activity` method `onOptionsItemSelected`. (Part 2 of 2.)

5.5.6 Anonymous Inner Class That Implements `OnSharedPreferenceChangeListener`

The `preferenceChangeListener` (Fig. 5.23) is an anonymous-inner-class object that implements the `OnSharedPreferenceChangeListener` interface. This object was registered in method `onCreate` to listen for changes to the app's `SharedPreferences`. When a change occurs, method `onSharedPreferenceChanged` sets `preferencesChanged` to `true` (line 120), then gets a reference to the `QuizFragment` (lines 122–123) so that the quiz can be reset with the new preferences. If the `CHOICES` preference changed, lines 127–128 call the `QuizFragment`'s `updateGuessRows` and `resetQuiz` methods.

```
111       // listener for changes to the app's SharedPreferences
112       private OnSharedPreferenceChangeListener preferenceChangeListener =
113          new OnSharedPreferenceChangeListener()
114       {
115          // called when the user changes the app's preferences
116          @Override
117          public void onSharedPreferenceChanged(
118             SharedPreferences sharedPreferences, String key)
119          {
120             preferencesChanged = true; // user changed app settings
121
122             QuizFragment quizFragment = (QuizFragment)
123                getFragmentManager().findFragmentById(R.id.quizFragment);
124
125             if (key.equals(CHOICES)) // # of choices to display changed
126             {
127                quizFragment.updateGuessRows(sharedPreferences);
128                quizFragment.resetQuiz();
129             }
130             else if (key.equals(REGIONS)) // regions to include changed
131             {
132                Set<String> regions =
133                   sharedPreferences.getStringSet(REGIONS, null);
134
135                if (regions != null && regions.size() > 0)
136                {
137                   quizFragment.updateRegions(sharedPreferences);
138                   quizFragment.resetQuiz();
139                }
140                else // must select one region--set North America as default
141                {
```

Fig. 5.23 | Anonymous Inner class that implements `OnSharedPreferenceChangeListener`. (Part 1 of 2.)

```
142               SharedPreferences.Editor editor = sharedPreferences.edit();
143               regions.add(
144                  getResources().getString(R.string.default_region));
145               editor.putStringSet(REGIONS, regions);
146               editor.commit();
147               Toast.makeText(MainActivity.this,
148                  R.string.default_region_message,
149                  Toast.LENGTH_SHORT).show();
150            }
151         }
152
153         Toast.makeText(MainActivity.this,
154            R.string.restarting_quiz, Toast.LENGTH_SHORT).show();
155      } // end method onSharedPreferenceChanged
156    }; // end anonymous inner class
157 } // end class MainActivity
```

Fig. 5.23 | Anonymous Inner class that implements `OnSharedPreferenceChangeListener`. (Part 2 of 2.)

If the `REGIONS` preference changed, lines 132–133 get the `Set<String>` containing the enabled regions. `SharedPreferences` method **`getStringSet`** returns a `Set<String>` for the specified key. The quiz must have at least one region enabled, so if the `Set<String>` is not empty, lines 137–138 call the `QuizFragment`'s `updateRegions` and `resetQuiz` methods. Otherwise, lines 142–146 update the `REGIONS` preference with North America set as the default region, and lines 147–149 use a `Toast` to indicate that the default region was set. `Toast` method **`makeText`** receives as arguments the `Context` on which the `Toast` is displayed, the message to display and the duration for which the `Toast` will be displayed. `Toast` method `show` displays the `Toast`. Regardless of which preference changed, lines 153–154 display a `Toast` indicating that the quiz will be reset with the new preferences. Figure 5.24 shows the `Toast` that appears after the user changes the app's preferences.

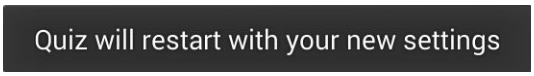

Quiz will restart with your new settings

Fig. 5.24 | `Toast` displayed after a preference is changed.

5.6 `QuizFragment` Class

Class `QuizFragment` (Figs. 5.25–5.34) builds the **Flag Quiz**'s GUI and implements the quiz's logic.

5.6.1 package Statement and `import` Statements

Figure 5.25 shows the `QuizFragment` `package` statement and `import` statements. Lines 5–33 `import` the various Java and Android classes and interfaces that the app uses. We've highlighted the new `import` statements, and we discuss the corresponding classes and interfaces in Section 5.3 and as they're encountered in Sections 5.6.2—5.6.10.

```
 1 // QuizFragment.java
 2 // Contains the Flag Quiz logic
 3 package com.deitel.flagquiz;
 4
 5 import java.io.IOException;
 6 import java.io.InputStream;
 7 import java.security.SecureRandom;
 8 import java.util.ArrayList;
 9 import java.util.Collections;
10 import java.util.List;
11 import java.util.Set;
12
13 import android.app.AlertDialog;
14 import android.app.Dialog;
15 import android.app.DialogFragment;
16 import android.app.Fragment;
17 import android.content.DialogInterface;
18 import android.content.SharedPreferences;
19 import android.content.res.AssetManager;
20 import android.graphics.drawable.Drawable;
21 import android.os.Bundle;
22 import android.os.Handler;
23 import android.util.Log;
24 import android.view.LayoutInflater;
25 import android.view.View;
26 import android.view.View.OnClickListener;
27 import android.view.ViewGroup;
28 import android.view.animation.Animation;
29 import android.view.animation.AnimationUtils;
30 import android.widget.Button;
31 import android.widget.ImageView;
32 import android.widget.LinearLayout;
33 import android.widget.TextView;
34
```

Fig. 5.25 | QuizFragment package statement, import statements.

5.6.2 Fields

Figure 5.26 lists class QuizFragment's static and instance variables. The constant TAG (line 38) is used when we log error messages using class Log (Fig. 5.31) to distinguish this Activity's error messages from others that are being written to the device's log. The constant FLAGS_IN_QUIZ (line 40) represents the number of flags in the quiz.

```
35 public class QuizFragment extends Fragment
36 {
37    // String used when logging error messages
38    private static final String TAG = "FlagQuiz Activity";
39
40    private static final int FLAGS_IN_QUIZ = 10;
41
```

Fig. 5.26 | QuizFragment fields. (Part 1 of 2.)

```
42      private List<String> fileNameList; // flag file names
43      private List<String> quizCountriesList; // countries in current quiz
44      private Set<String> regionsSet; // world regions in current quiz
45      private String correctAnswer; // correct country for the current flag
46      private int totalGuesses; // number of guesses made
47      private int correctAnswers; // number of correct guesses
48      private int guessRows; // number of rows displaying guess Buttons
49      private SecureRandom random; // used to randomize the quiz
50      private Handler handler; // used to delay loading next flag
51      private Animation shakeAnimation; // animation for incorrect guess
52
53      private TextView questionNumberTextView; // shows current question #
54      private ImageView flagImageView; // displays a flag
55      private LinearLayout[] guessLinearLayouts; // rows of answer Buttons
56      private TextView answerTextView; // displays Correct! or Incorrect!
57
```

Fig. 5.26 | `QuizFragment` fields. (Part 2 of 2.)

Variable `fileNameList` (line 42) holds the flag image file names for the currently enabled geographic regions. Variable `quizCountriesList` (line 43) holds the flag file names for the countries used in the current quiz. Variable `regionsSet` (line 44) stores the geographic regions that are enabled.

Variable `correctAnswer` (line 45) holds the flag file name for the current flag's correct answer. Variable `totalGuesses` (line 46) stores the total number of correct and incorrect guesses the player has made so far. Variable `correctAnswers` (line 47) is the number of correct guesses so far; this will eventually be equal to `FLAGS_IN_QUIZ` if the user completes the quiz. Variable `guessRows` (line 48) is the number of three-`Button` `LinearLayouts` displaying the flag answer choices.

Variable `random` (line 49) is the random-number generator used to randomly pick the flags to include in the quiz and which `Button` in the three-`Button` `LinearLayouts` represents the correct answer. When the user selects a correct answer and the quiz is not over, we use the `Handler` object `handler` (line 50) to load the next flag after a short delay.

The `Animation` `shakeAnimation` (line 51) holds the dynamically inflated *shake animation* that's applied to the flag image when an incorrect guess is made. Lines 53–56 contain variables that we use to manipulate various GUI components programmatically.

5.6.3 Overridden Fragment Method onCreateView

`QuizFragment`'s `onCreateView` method (Fig. 5.27) inflates the GUI and initializes most of the `QuizFragment`'s instance variables—`guessRows` and `regionsSet` are initialized when the `MainActivity` calls `QuizFragment`'s `updateGuessRows` and `updateRegions` methods. After calling the superclass's `onCreateView` method (line 63), we inflate the `QuizFragment`'s GUI (line 64–65) using the `LayoutInflater` that method `onCreateView` receives as an argument. The `LayoutInflater`'s **`inflate`** method receives three arguments:

- the layout resource ID indicating the layout to inflate
- the `ViewGroup` (layout object) in which the `Fragment` will be displayed, which is received as `onCreateView`'s second argument

- a boolean indicating whether or not the inflated GUI needs to be attached to the ViewGroup in the second argument—false means that the Fragment was declared in the parent Activity's layout and true indicates that you're dynamically creating the Fragment and its GUI should be attached.

Method inflate returns a reference to a View that contains the inflated GUI. We store that in local variable view so that it can be returned by onCreateView after the QuizFragment's other instance variables are initialized.

```java
   // configures the QuizFragment when its View is created
   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
      Bundle savedInstanceState)
   {
      super.onCreateView(inflater, container, savedInstanceState);
      View view =
         inflater.inflate(R.layout.fragment_quiz, container, false);

      fileNameList = new ArrayList<String>();
      quizCountriesList = new ArrayList<String>();
      random = new SecureRandom();
      handler = new Handler();

      // load the shake animation that's used for incorrect answers
      shakeAnimation = AnimationUtils.loadAnimation(getActivity(),
         R.anim.incorrect_shake);
      shakeAnimation.setRepeatCount(3); // animation repeats 3 times

      // get references to GUI components
      questionNumberTextView =
         (TextView) view.findViewById(R.id.questionNumberTextView);
      flagImageView = (ImageView) view.findViewById(R.id.flagImageView);
      guessLinearLayouts = new LinearLayout[3];
      guessLinearLayouts[0] =
         (LinearLayout) view.findViewById(R.id.row1LinearLayout);
      guessLinearLayouts[1] =
         (LinearLayout) view.findViewById(R.id.row2LinearLayout);
      guessLinearLayouts[2] =
         (LinearLayout) view.findViewById(R.id.row3LinearLayout);
      answerTextView = (TextView) view.findViewById(R.id.answerTextView);

      // configure listeners for the guess Buttons
      for (LinearLayout row : guessLinearLayouts)
      {
         for (int column = 0; column < row.getChildCount(); column++)
         {
            Button button = (Button) row.getChildAt(column);
            button.setOnClickListener(guessButtonListener);
         }
      }

```

Fig. 5.27 | QuizFragment overridden Fragment method onCreateView. (Part 1 of 2.)

```
100         // set questionNumberTextView's text
101         questionNumberTextView.setText(
102            getResources().getString(R.string.question, 1, FLAGS_IN_QUIZ));
103         return view; // returns the fragment's view for display
104      } // end method onCreateView
105
```

Fig. 5.27 | `QuizFragment` overridden `Fragment` method `onCreateView`. (Part 2 of 2.)

Lines 67–68 create `ArrayList<String>` objects that will store the flag image file names for the currently enabled geographical regions and the names of the countries in the current quiz, respectively. Line 69 creates the `SecureRandom` object for randomizing the quiz's flags and guess `Buttons`. Line 70 creates the `Handler` object `handler`, which we'll use to delay by two seconds the appearance of the next flag after the user correctly guesses the current flag.

Lines 73–74 dynamically load the *shake animation* that will be applied to the flag when an incorrect guess is made. `AnimationUtils` `static` method `loadAnimation` loads the animation from the XML file represented by the constant `R.anim.incorrect_shake`. The first argument indicates the `Context` containing the resources that will be animated—inherited `Fragment` method `getActivity` returns the `Activity` that hosts this `Fragment`. `Activity` is an indirect subclass of `Context`. Line 75 specifies the number of times the animation should repeat with `Animation` method `setRepeatCount`.

Lines 78–88 get references to various GUI components that we'll programmatically manipulate. Lines 91–98 get each guess `Button` from the three `guessLinearLayouts` and register `guessButtonListener` (Section 5.6.9) as the `OnClickListener`.

Lines 101–102 set the text in `questionNumberTextView` to the `String` returned by `String` `static` method `format`. The first argument to `format` is the `String` resource `R.string.question`, which is a format `String` containing placeholders for two integer values (as described in Section 5.4.2). Inherited `Fragment` method **`getResources`** returns a `Resources` object (package `android.content.res`) that can be used to load resources. We then call that object's **`getString`** method to load the `R.string.question` resource, which represents the `String`

```
Question %1$d of %2$d
```

Line 103 returns the `QuizFragment`'s GUI.

5.6.4 Method updateGuessRows

Method `updateGuessRows` (Fig. 5.28) is called from the app's `MainActivity` when the app is launched and each time the user changes the number of guess `Buttons` to display with each flag. Lines 110–111 use the method's `SharedPreferences` argument to get the `String` for the key `MainActivity.CHOICES`—a constant containing the name of the preference in which the `SettingsFragment` stores the number of guess `Buttons` to display. Line 112 converts the preference's value to an `int` and divides it by 3 to determine the value for `guessRows`, which indicates how many of the `guessLinearLayouts` should be displayed—each with three guess `Buttons`. Next, lines 115–116 hide all of the `guessLinearLayouts`, so that lines 119–120 can show the appropriate `guessLinearLayouts` based on the value of `guessRows`.

```
106    // update guessRows based on value in SharedPreferences
107    public void updateGuessRows(SharedPreferences sharedPreferences)
108    {
109       // get the number of guess buttons that should be displayed
110       String choices =
111          sharedPreferences.getString(MainActivity.CHOICES, null);
112       guessRows = Integer.parseInt(choices) / 3;
113
114       // hide all guess button LinearLayouts
115       for (LinearLayout layout : guessLinearLayouts)
116          layout.setVisibility(View.INVISIBLE);
117
118       // display appropriate guess button LinearLayouts
119       for (int row = 0; row < guessRows; row++)
120          guessLinearLayouts[row].setVisibility(View.VISIBLE);
121    }
122
```

Fig. 5.28 | `QuizFragment` method `updateGuessRows`.

5.6.5 Method updateRegions

Method `updateRegions` (Fig. 5.29) is called from the app's `MainActivity` when the app is launched and each time the user changes the world regions that should be included in the quiz. Lines 126–127 use the method's `SharedPreferences` argument to get the names of all of the enabled regions as a `Set<String>`. `MainActivity.REGIONS` is a constant containing the name of the preference in which the `SettingsFragment` stores the enabled world regions.

```
123    // update world regions for quiz based on values in SharedPreferences
124    public void updateRegions(SharedPreferences sharedPreferences)
125    {
126       regionsSet =
127          sharedPreferences.getStringSet(MainActivity.REGIONS, null);
128    }
129
```

Fig. 5.29 | `QuizFragment` method `updateRegions`.

5.6.6 Method resetQuiz

Method `resetQuiz` (Fig. 5.30) sets up and starts a quiz. Recall that the images for the game are stored in the app's `assets` folder. To access this folder's contents, the method gets the app's `AssetManager` (line 134) by calling the parent `Activity`'s **`getAssets`** method. Next, line 135 clears the `fileNameList` to prepare to load image file names for only the enabled geographical regions. Lines 140–147 iterate through all the enabled world regions. For each, we use the `AssetManager`'s `list` method (line 143) to get an array of the flag image file names, which we store in the `String` array `paths`. Lines 145–146 remove the `.png` extension from each file name and place the names in the `fileNameList`. `AssetManager`'s `list` method throws `IOException`s, which are *checked* exceptions (so you must catch or declare the exception). If an exception occurs because the app is unable to access the `assets` folder, lines 149–152 catch the exception and *log* it for debugging purposes with Android's built-in logging mechanism. `Log` `static` method **`e`** is used to log error messages. You can see the complete list of `Log` methods at

```
http://developer.android.com/reference/android/util/Log.html
```

```
130    // set up and start the next quiz
131    public void resetQuiz()
132    {
133       // use AssetManager to get image file names for enabled regions
134       AssetManager assets = getActivity().getAssets();
135       fileNameList.clear(); // empty list of image file names
136
137       try
138       {
139          // loop through each region
140          for (String region : regionsSet)
141          {
142             // get a list of all flag image files in this region
143             String[] paths = assets.list(region);
144
145             for (String path : paths)
146                fileNameList.add(path.replace(".png", ""));
147          }
148       }
149       catch (IOException exception)
150       {
151          Log.e(TAG, "Error loading image file names", exception);
152       }
153
154       correctAnswers = 0; // reset the number of correct answers made
155       totalGuesses = 0; // reset the total number of guesses the user made
156       quizCountriesList.clear(); // clear prior list of quiz countries
157
158       int flagCounter = 1;
159       int numberOfFlags = fileNameList.size();
160
161       // add FLAGS_IN_QUIZ random file names to the quizCountriesList
162       while (flagCounter <= FLAGS_IN_QUIZ)
163       {
164          int randomIndex = random.nextInt(numberOfFlags);
165
166          // get the random file name
167          String fileName = fileNameList.get(randomIndex);
168
169          // if the region is enabled and it hasn't already been chosen
170          if (!quizCountriesList.contains(fileName))
171          {
172             quizCountriesList.add(fileName); // add the file to the list
173             ++flagCounter;
174          }
175       }
```

Fig. 5.30 | `QuizFragment` method `resetQuiz`. (Part 1 of 2.)

```
176
177       loadNextFlag(); // start the quiz by loading the first flag
178    } // end method resetQuiz
179
```

Fig. 5.30 | `QuizFragment` method `resetQuiz`. (Part 2 of 2.)

Next, lines 154–156 reset the counters for the number of correct guesses the user has made (`correctAnswers`) and the total number of guesses the user has made (`totalGuesses`) to 0 and clear the `quizCountriesList`.

Lines 162–175 add `FLAGS_IN_QUIZ` (10) randomly selected file names to the `quizCountriesList`. We get the total number of flags, then randomly generate the index in the range 0 to one less than the number of flags. We use this index to select one image file name from `fileNamesList`. If the `quizCountriesList` does not already contain that file name, we add it to `quizCountriesList` and increment the `flagCounter`. We repeat this process until `FLAGS_IN_QUIZ` unique file names have been selected. Then line 177 calls `loadNextFlag` (Fig. 5.31) to load the quiz's first flag.

5.6.7 Method `loadNextFlag`

Method `loadNextFlag` (Fig. 5.31) loads and displays the next flag and the corresponding set of answer `Button`s. The image file names in `quizCountriesList` have the format

regionName-countryName

without the `.png` extension. If a *regionName* or *countryName* contains multiple words, they're separated by underscores (_).

```
180    // after the user guesses a correct flag, load the next flag
181    private void loadNextFlag()
182    {
183       // get file name of the next flag and remove it from the list
184       String nextImage = quizCountriesList.remove(0);
185       correctAnswer = nextImage; // update the correct answer
186       answerTextView.setText(""); // clear answerTextView
187
188       // display current question number
189       questionNumberTextView.setText(
190          getResources().getString(R.string.question,
191             (correctAnswers + 1), FLAGS_IN_QUIZ));
192
193       // extract the region from the next image's name
194       String region = nextImage.substring(0, nextImage.indexOf('-'));
195
196       // use AssetManager to load next image from assets folder
197       AssetManager assets = getActivity().getAssets();
198
199       try
200       {
```

Fig. 5.31 | `QuizFragment` method `loadNextFlag`. (Part 1 of 2.)

```
         // get an InputStream to the asset representing the next flag
         InputStream stream =
            assets.open(region + "/" + nextImage + ".png");

         // load the asset as a Drawable and display on the flagImageView
         Drawable flag = Drawable.createFromStream(stream, nextImage);
         flagImageView.setImageDrawable(flag);
      }
      catch (IOException exception)
      {
         Log.e(TAG, "Error loading " + nextImage, exception);
      }

      Collections.shuffle(fileNameList); // shuffle file names

      // put the correct answer at the end of fileNameList
      int correct = fileNameList.indexOf(correctAnswer);
      fileNameList.add(fileNameList.remove(correct));

      // add 3, 6, or 9 guess Buttons based on the value of guessRows
      for (int row = 0; row < guessRows; row++)
      {
         // place Buttons in currentTableRow
         for (int column = 0;
            column < guessLinearLayouts[row].getChildCount(); column++)
         {
            // get reference to Button to configure
            Button newGuessButton =
               (Button) guessLinearLayouts[row].getChildAt(column);
            newGuessButton.setEnabled(true);

            // get country name and set it as newGuessButton's text
            String fileName = fileNameList.get((row * 3) + column);
            newGuessButton.setText(getCountryName(fileName));
         }
      }

      // randomly replace one Button with the correct answer
      int row = random.nextInt(guessRows); // pick random row
      int column = random.nextInt(3); // pick random column
      LinearLayout randomRow = guessLinearLayouts[row]; // get the row
      String countryName = getCountryName(correctAnswer);
      ((Button) randomRow.getChildAt(column)).setText(countryName);
   } // end method loadNextFlag

```

Fig. 5.31 | `QuizFragment` method `loadNextFlag`. (Part 2 of 2.)

Line 184 removes the first name from `quizCountriesList` and stores it in `nextImage`. We also save this in `correctAnswer` so it can be used later to determine whether the user made a correct guess. Next, we clear the `answerTextView` and display the current question number in the `questionNumberTextView` (lines 189–191) using the formatted `String` resource `R.string.question`.

Line 194 extracts from nextImage the region to be used as the assets subfolder name from which we'll load the image. Next we get the AssetManager, then use it in the try statement to open an InputStream (package java.io) to read bytes from the flag image's file. We use that stream as an argument to class **Drawable**'s static method **createFromStream**, which creates a Drawable object (package android.graphics.drawable). The Drawable is set as flagImageView's item to display by calling its **setImageDrawable** method. If an exception occurs, we log it for debugging purposes (line 211).

Next, line 214 shuffles the fileNameList, and lines 217–218 locate the correctAnswer and move it to the end of the fileNameList—later we'll insert this answer randomly into the one of the guess Buttons.

Lines 221–236 iterate through the Buttons in the guessLinearLayouts for the current number of guessRows. For each Button:

- lines 228–229 get a reference to the next Button
- line 230 enables the Button
- line 233 gets the flag file name from the fileNameList
- line 234 sets Button's text with the country name that's returned by method getCountryName (Section 5.6.8)

Lines 239–243 pick a random row (based on the current number of guessRows) and column, then set the text of the corresponding Button.

5.6.8 Method getCountryName

Method getCountryName (Fig. 5.32) parses the country name from the image file name. First, we get a substring starting from the dash (-) that separates the region from the country name. Then we call String method replace to replace the underscores (_) with spaces.

```
   // parses the country flag file name and returns the country name
   private String getCountryName(String name)
   {
      return name.substring(name.indexOf('-') + 1).replace('_', ' ');
   }

```

Fig. 5.32 | QuizFragment method getCountryName.

5.6.9 Anonymous Inner Class That Implements OnClickListener

Lines 91–98 (Fig. 5.27) registered guessButtonListener (Fig. 5.33) as the event-handling object for each guess Button. Instance variable guessButtonListener refers to an anonymous inner class object that implements interface OnClickListener to respond to Button events. The method receives the clicked Button as parameter v. We get the Button's text (line 259) and the parsed country name (line 260), then increment totalGuesses.

If the guess is correct (line 263), we increment correctAnswers. Next, we set the answerTextView's text to the country name and change its color to the color represented by the constant R.color.correct_answer (green), and we call our utility method disableButtons (Section 5.6.10) to disable all the answer Buttons.

```
   // called when a guess Button is touched
   private OnClickListener guessButtonListener = new OnClickListener()
   {
      @Override
      public void onClick(View v)
      {
         Button guessButton = ((Button) v);
         String guess = guessButton.getText().toString();
         String answer = getCountryName(correctAnswer);
         ++totalGuesses; // increment number of guesses the user has made

         if (guess.equals(answer)) // if the guess is correct
         {
            ++correctAnswers; // increment the number of correct answers

            // display correct answer in green text
            answerTextView.setText(answer + "!");
            answerTextView.setTextColor(
               getResources().getColor(R.color.correct_answer));

            disableButtons(); // disable all guess Buttons

            // if the user has correctly identified FLAGS_IN_QUIZ flags
            if (correctAnswers == FLAGS_IN_QUIZ)
            {
               // DialogFragment to display quiz stats and start new quiz
               DialogFragment quizResults =
                  new DialogFragment()
                  {
                     // create an AlertDialog and return it
                     @Override
                     public Dialog onCreateDialog(Bundle bundle)
                     {
                        AlertDialog.Builder builder =
                           new AlertDialog.Builder(getActivity());
                        builder.setCancelable(false);

                        builder.setMessage(
                           getResources().getString(R.string.results,
                           totalGuesses, (1000 / (double) totalGuesses)));

                        // "Reset Quiz" Button
                        builder.setPositiveButton(R.string.reset_quiz,
                           new DialogInterface.OnClickListener()
                           {
                              public void onClick(DialogInterface dialog,
                                 int id)
                              {
                                 resetQuiz();
                              }
                           } // end anonymous inner class
                        ); // end call to setPositiveButton
```

Fig. 5.33 | Anonymous inner class that implements `OnClickListener`. (Part 1 of 2.)

```
304
305                           return builder.create(); // return the AlertDialog
306                        } // end method onCreateDialog
307                     }; // end DialogFragment anonymous inner class
308
309                  // use FragmentManager to display the DialogFragment
310                  quizResults.show(getFragmentManager(), "quiz results");
311               }
312               else // answer is correct but quiz is not over
313               {
314                  // load the next flag after a 1-second delay
315                  handler.postDelayed(
316                     new Runnable()
317                     {
318                        @Override
319                        public void run()
320                        {
321                           loadNextFlag();
322                        }
323                     }, 2000); // 2000 milliseconds for 2-second delay
324               }
325            }
326            else // guess was incorrect
327            {
328               flagImageView.startAnimation(shakeAnimation); // play shake
329
330               // display "Incorrect!" in red
331               answerTextView.setText(R.string.incorrect_answer);
332               answerTextView.setTextColor(
333                  getResources().getColor(R.color.incorrect_answer));
334               guessButton.setEnabled(false); // disable incorrect answer
335            }
336         }
337      }; // end guessButtonListener
338
```

Fig. 5.33 | Anonymous inner class that implements `OnClickListener`. (Part 2 of 2.)

If `correctAnswers` is `FLAGS_IN_QUIZ` (line 275), the quiz is over. Lines 278–307 create a new anonymous inner class that extends `DialogFragment` and will be used to display the quiz results. The `DialogFragment`'s **`onCreateDialog`** method uses an `AlertDialog.Builder` to configure and create an `AlertDialog`, then returns it. When the user touches the dialog's **Reset Quiz** `Button`, method `resetQuiz` is called to start a new game (line 300). To display the `DialogFragment`, line 310 calls its **`show`** method, passing as arguments the `FragmentManager` returned by `getFragmentManager` and a `String`. The second argument can be used with `FragmentManager` method **`getFragmentByTag`** to get a reference to the `DialogFragment` at a later time—we don't use this capability in this app.

If `correctAnswers` is less than `FLAGS_IN_QUIZ`, then lines 315–323 call the `postDelayed` method of `Handler` object `handler`. The first argument defines an anonymous inner class that implements the `Runnable` interface—this represents the task to perform (`loadNextFlag`) some number of milliseconds into the future. The second argument is the delay in milliseconds (`2000`). If the guess is incorrect, line 328 invokes `flagImageView`'s `start-`

`Animation` method to play the `shakeAnimation` that was loaded in method `onCreateView`. We also set the text on `answerTextView` to display `"Incorrect!"` in red (lines 331–333), then disable the `guessButton` that corresponds to the incorrect answer.

5.6.10 Method `disableButtons`

Method `disableButtons` (Fig. 5.34) iterates through the guess `Buttons` and disables them.

```java
   // utility method that disables all answer Buttons
   private void disableButtons()
   {
      for (int row = 0; row < guessRows; row++)
      {
         LinearLayout guessRow = guessLinearLayouts[row];
         for (int i = 0; i < guessRow.getChildCount(); i++)
            guessRow.getChildAt(i).setEnabled(false);
      }
   }
} // end class FlagQuiz
```

Fig. 5.34 | `QuizFragment` method `disableButtons`.

5.7 `SettingsFragment` Class

Class `SettingsFragment` (Fig. 5.35) extends `PreferenceFragment`, which provides capabilities for managing the app's settings. Overridden method `onCreate` (lines 11–16) is called when the `SettingsFragment` is created—either by the `SettingsActivity` when the app is running in portrait orientation or by the `MainActivity` when the app is running on a tablet in landscape orientation. Line 15 uses inherited `PreferenceFragment` method **`addPreferencesFromResource`** to build the preferences GUI. The argument is the resource ID for the `preferences.xml` file you created in Section 5.4.10.

```java
 1  // SettingsFragment.java
 2  // Subclass of PreferenceFragment for managing app settings
 3  package com.deitel.flagquiz;
 4
 5  import android.os.Bundle;
 6  import android.preference.PreferenceFragment;
 7
 8  public class SettingsFragment extends PreferenceFragment
 9  {
10     // creates preferences GUI from preferences.xml file in res/xml
11     @Override
12     public void onCreate(Bundle savedInstanceState)
13     {
14        super.onCreate(savedInstanceState);
15        addPreferencesFromResource(R.xml.preferences); // load from XML
16     }
17  } // end class SettingsFragment
```

Fig. 5.35 | Subclass of `PreferenceFragment` for managing app settings.

5.8 SettingsActivity Class

Class SettingsActivity (Fig. 5.36) hosts the SettingsFragment when the app is running in portrait orientation. To create this class, right click the package (com.deitel.flagquiz) and select **New > Class** to display the **New Java Class** dialog. Set the new class's **Name** to SettingsActivity, set its **Superclass** to android.app.Activity and click **Finish**.

Overridden method onCreate (lines 11–16) calls Activity method setContentView to inflate the GUI defined by activity_settings.xml (Section 5.4.6)—represented by the resource R.layout.activity_settings.

```
 1  // SettingsActivity.java
 2  // Activity to display SettingsFragment on a phone
 3  package com.deitel.flagquiz;
 4
 5  import android.app.Activity;
 6  import android.os.Bundle;
 7
 8  public class SettingsActivity extends Activity
 9  {
10     // use FragmentManager to display SettingsFragment
11     @Override
12     protected void onCreate(Bundle savedInstanceState)
13     {
14        super.onCreate(savedInstanceState);
15        setContentView(R.layout.activity_settings);
16     }
17  } // end class SettingsActivity
```

Fig. 5.36 | Activity to display SettingsFragment on a phone.

5.9 AndroidManifest.xml

Each Activity in an app must be declared in the app's AndroidManifest.xml file; otherwise, Android will not know that the Activity exists and will not be able to launch it. When you created the app, the IDE declared its MainActivity in AndroidManifest.xml. To declare the app's SettingsActivity:

1. Open AndroidManifest.xml and click the **Application** tab at the bottom of the manifest editor.
2. In the **Application Nodes** section, click **Add...**, select **Activity** from the dialog that appears and click **OK**.
3. In the **Application Nodes** section, select the new **Activity** node to display its attributes in the **Attributes for Activity** section.
4. In the **Name** field, enter .SettingsActivity. The dot (.) before SettingsActivity is shorthand notation for the app's package name (com.deitel.flagquiz).
5. In the **Label** field, enter @string/settings_activity—this string resource is displayed in the action bar when the SettingsActivity is running.

For complete manifest file details, visit http://developer.android.com/guide/topics/manifest/manifest-intro.html.

5.10 Wrap-Up

In this chapter, you built a **Flag Quiz** app that tests a user's ability to correctly identify country flags. A key feature of this chapter was using `Fragments` to create portions of an `Activity`'s GUI. You used two activities to display the `QuizFragment` and the `SettingsFragment` when the app was running in portrait orientation, and one `Activity` to display both `Fragments` when the app was running on a tablet in landscape orientation—thus, making better use of the available screen real estate. You used a subclass of `PreferenceFragment` to automatically maintain and persist the app's settings and a subclass of `DialogFragment` to display an `AlertDialog` to the user. We discussed portions of a `Fragment`'s lifecycle and showed how to use the `FragmentManager` to obtain a reference to a `Fragment` so that you could interact with it programmatically.

In portrait orientation, you used the app's action menu to enable the user to display the `SettingsActivity` containing the `SettingsFragment`. To launch the `SettingsActivity`, you used an explicit `Intent`.

We showed how to use Android's `WindowManager` to obtain a `Display` object so that you could determine whether the app was running on a tablet in landscape orientation. In this case, you prevented the menu from displaying because the `SettingsFragment` was already on the screen.

We demonstrated how to manage a large number of image resources using subfolders in the app's assets folder and how to access those resources via an `AssetManager`. You created a `Drawable` from an image's bytes by reading them from an `InputStream`, then displayed the `Drawable` in an `ImageView`.

You learned about additional subfolders of the app's `res` folder—`menu` for storing menu resource files, `anim` for storing animation resource files and `xml` for storing raw XML data files. We also discussed how to use qualifiers to create a folder for storing a layout that should be used only on large devices in landscape orientation.

You used `Toasts` to briefly display minor error or informational messages. To display the next flag in the quiz after a short delay, you used a `Handler`, which executes a `Runnable` after a specified number of milliseconds. You learned that a `Handler`'s `Runnable` executes in the thread that created the `Handler` (the GUI thread in this app).

We defined an `Animation` in XML and applied it to the app's `ImageView` when the user guessed incorrectly to provide visual feedback to the user. You learned how to log exceptions for debugging purposes with Android's built-in logging mechanism and class `Log`. You also used additional classes and interfaces from the `java.util` package, including `List`, `ArrayList`, `Collections` and `Set`.

In Chapter 6, you'll create a **Cannon Game** using multithreading and frame-by-frame animation. You'll handle touch gestures to fire a cannon. You'll learn how to create a game loop that updates the display as fast as possible to create smooth animations and to make the game feel like it executes at the same speed regardless of a given device's processor speed. We'll also show how to perform simple collision detection.

Self-Review Exercises

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

a) `FragmentManager` can use __________ class of package `android.app` to dynamically add, remove and transition between fragments.

b) Files in the `assets` folders are accessed via a(n) __________ (package `android.content.res`), which can provide a list of all of the file names in a specified subfolder of `assets` and can be used to access each asset.
c) __________ method of `PreferenceManager` returns a reference to the `SharedPreferences` object.
d) By default, animations in an animation `set` are applied in parallel, but you can use the __________ attribute to specify the number of milliseconds into the future at which an animation should begin. This can be used to sequence the animations in a `set`.

5.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) We use `AnimationUtils static` method `loadAnimation` to load an animation from an XML file that specifies the animation's options.
b) Android does not provide a logging mechanism for debugging purposes.
c) `ImageView`'s **Adjust View Bounds** property specifies whether or not the `ImageView` maintains the aspect ratio of its `Drawable`.
d) You load color and `String` array resources from the `colors.xml` and `strings.xml` files into memory by using the `Activity`'s `Resources` object.
e) Use activities to create reusable components and make better use of the screen real estate in a tablet app.

Answers to Self-Review Exercises

5.1 a) `FragmentTransaction`. b) `AssetManager`. c) `getDefaultSharedPreferences`. d) `android:startOffset`.

5.2 a) True. b) False. When exceptions occur, you can log them for debugging purposes with the built-in `Log` class's methods. c) True. d) True. e) False. Use `Fragments` to create reusable components and make better use of the screen real estate in a tablet app.

Exercises

5.3 Fill in the blanks in each of the following statements:
a) To specify `Menu` options, you override Activity's __________ method to add the options to the method's `Menu` argument.
b) To delay an action, we use a(n) __________ (package `android.os`) object to execute a `Runnable` after a specified delay.
c) You can specify the number of times an animation should repeat with `Animation` method __________ and perform the animation by calling `View` method `startAnimation` (with the `Animation` as an argument) on the `ImageView`.
d) __________ method of `Fragment` builds and returns a `View` containing the `Fragment`'s GUI.
e) Android supports __________ animations which allow you to animate any property of any object.
f) For the `android:fromXDelta` attribute, specifying the value `-5%p` indicates that the `View` should move to the __________ by 5% of the parent's width (indicated by the `p`).
g) We use the __________ attribute of the `application` element to apply a theme to the application's GUI.

5.4 State whether each of the following is *true* or *false*. If *false*, explain why.
a) Resource folder names that begin with `anim` contain XML files that define tweened animations.
b) One would use Android's `DisplayManager` to obtain a `Display` object that contains the screen size.
c) `Fragments` can be executed independently of a parent `Activity`.

Project Exercises

5.5 *(Enhanced* **Flag Quiz** *App)* Make the following enhancements to the **Flag Quiz** app:

a) Count the number of questions that were answered correctly on the first try. After all the questions have been answered, display a message describes how well the user performed on first guesses.
b) Keep track of the score as the user proceeds through the app. Give the user the most points for answering correctly on the first guess, fewer points for answering correctly on the next guess, etc.
c) Use a `SharedPreferences` file to save the top five high scores.
d) Add multiplayer functionality.
e) If the user guesses the correct flag, include a "bonus question" asking the user to name the capital of the country. If the user answers correctly on the first guess, add 10 bonus points to the score; otherwise, simply display the correct answer, then allow the user to proceed to the next flag.
f) After the user answers the question correctly, include a link to the Wikipedia for that country so the user can learn more about the country as they play the game. In this version of the app, you may want to allow the user to decide when to move to the next flag.

5.6 *(Favorite Celebrities App with `Fragments`)* Reimplement the **Favorite Celebrities App** of Chapter 4 using a `Fragment`. Rather than having the `Activity` extend `ListActivity`, create a subclass of `ListFragment`, then host an object of your new subclass in the class's `MainActivity`.

5.7 *(Road Sign Quiz App)* Create an app that tests the user's knowledge of road signs. Display a random sign image and ask the user to select the sign's name. Visit `http://mutcd.fhwa.dot.gov/ser-shs_millennium.htm` for traffic sign images and information.

5.8 *(U.S. State Quiz App)* Using the techniques you learned in this chapter, create an app that displays an outline of a U.S. state and asks the user to identify the state. If the user guesses the correct state, include a "bonus question" asking the user to name the state's capital. If the user answers correctly, add 10 bonus points to the score; otherwise, simply display the correct answer, then allow the user to proceed to the next state. Keep score as described in Exercise 5.5(c).

5.9 *(Country Quiz App)* Using the techniques you learned in this chapter, create an app that displays an outline of a country and asks the user to identify its name. If the user guesses the correct country, include a "bonus question" asking the user to name the country's capital. If the user answers correctly, add 10 bonus points to the score; otherwise, simply display the correct answer, then allow the user to proceed to the next country. Keep score as described in Exercise 5.5(c).

5.10 *(Android Programming Quiz App)* Using the Android knowledge you've gained thus far, create a multiple-choice Android programming quiz *using original questions that you create*. Add multiplayer capabilities so you can compete against your classmates.

5.11 *(Cricket Trivia Quiz App)* Create a cricket trivia quiz app.

5.12 *(Science Quiz App)* Create a science quiz app.

5.13 *(Custom Quiz App)* Create an app that allows the user to create a customized true/false or multiple-choice quiz. This is a great study aid. The user can input questions on any subject and include answers, then use it to study for a test or final exam.

5.14 *(Lottery Number Picker App)* Create an app that randomly picks lottery numbers. Ask the user how many numbers to pick and the maximum valid number in the lottery (set a maximum value of 99). Provide five possible lottery-number combinations to choose from. Include a feature that allows the user to easily pick from a list of five popular lottery games. Find five of the most popular lottery games in your area and research how many numbers must be picked for a lottery ticket and the highest valid number. Allow the user to tap the name of the lottery game to pick random numbers for that game.

5.15 *(Craps Game App)* Create an app that simulates playing the dice game of craps. In this game, a player rolls two dice. Each die has six faces—we've provided die images with the book's examples. Each face contains one, two, three, four, five or six spots. After the dice have come to rest, the sum of the spots on the two top faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw (called "craps"), the player loses (the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes the player's "point." To win, a player must continue rolling the dice until the point value is rolled. The player loses by rolling a 7 before rolling the point.

5.16 *(Craps Game App Modification)* Modify the craps app to allow wagering. Initialize the variable `balance` to 1000 dollars. Prompt the player to enter a `wager`. Check that `wager` is less than or equal to `balance`, and if it's not, have the user reenter `wager` until a valid `wager` is entered. After a correct `wager` is entered, run one game of craps. If the player wins, increase `balance` by `wager` and display the new `balance`. If the player loses, decrease `balance` by `wager`, display the new `balance`, check whether `balance` has become zero and, if so, display the message `"Sorry. You busted!"`

5.17 *(Computer-Assisted Instruction App)* Create an app that will help an elementary school student learn multiplication. Select two positive one-digit integers. The app should then prompt the user with a question, such as

```
How much is 6 times 7?
```

The student then inputs the answer. Next, the app checks the student's answer. If it's correct, display one of the following messages:

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

and ask another question. If the answer is wrong, display one of the following messages:

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

and let the student try the same question repeatedly until the student gets it right. Enhance the app to ask addition, subtraction and multiplication questions.

6 Cannon Game App

Objectives

In this chapter you'll:

- Create a simple game app that's easy to code and fun to play.
- Create a custom `SurfaceView` subclass for displaying the game's graphics from a separate thread of execution.
- Draw graphics using `Paint`s and a `Canvas`.
- Override `View`'s `onTouchEvent` method to fire a cannonball when the user touches the screen.
- Perform simple collision detection.
- Add sound to your app using a `SoundPool` and the `AudioManager`.
- Override `Fragment` lifecycle methods `onPause` and `onDestroy`.

Outline

6.1 Introduction

The **Cannon Game** app challenges you to destroy a seven-piece target before a ten-second time limit expires (Fig. 6.1). The game consists of four visual components—a *cannon* that you control, a *cannonball*, the *target* and a *blocker* that defends the target. You aim and fire the cannon by *touching* the screen—the cannon then aims at the touched point and fires the cannonball in a straight line in that direction. At the end of the game, the app displays an `AlertDialog` indicating whether you won or lost, and showing the number of shots fired and the elapsed time (Fig. 6.2).

The game begins with a *10-second time limit*. Each time you destroy a target section, a three-second time bonus is *added* to your remaining time, and each time you hit the blocker, a two-second time penalty is *subtracted* from your remaining time. You win by destroying all seven target sections before you run out of time—if the timer reaches zero, you lose.

When you fire the cannon, the game plays a *firing sound*. When a cannonball hits a target piece, a *glass-breaking sound* plays and that piece of the target disappears. When the cannonball hits the blocker, a *hit sound* plays and the cannonball bounces back. The blocker cannot be destroyed. The target and blocker move *vertically* at different speeds, changing direction when they hit the top or bottom of the screen.

[*Note:* Due to performance issues with the Android Emulator, you should test this app on an Android device.]

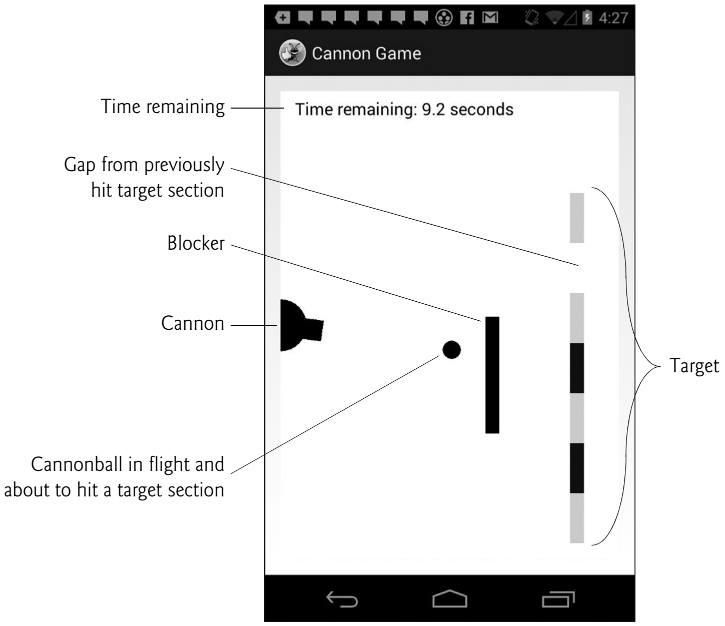
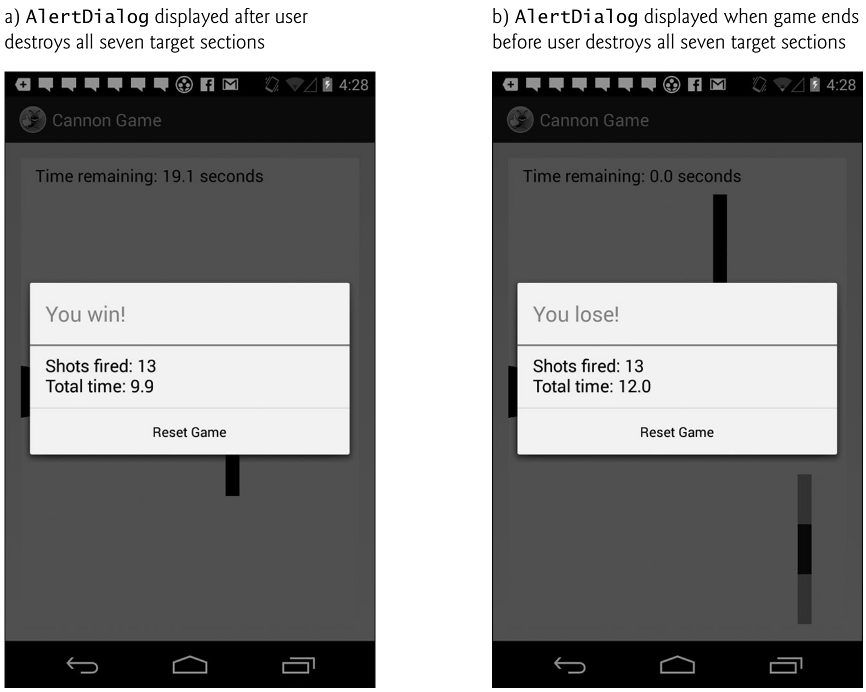

Time remaining
Gap from previously hit target section
Blocker
Cannon
Cannonball in flight and about to hit a target section
Target
Cannon Game
Time remaining: 9.2 seconds
4:27

Fig. 6.1 | Completed **Cannon Game** app.


a) AlertDialog displayed after user destroys all seven target sections
b) AlertDialog displayed when game ends before user destroys all seven target sections
Cannon Game
4:28
Time remaining: 19.1 seconds
Time remaining: 0.0 seconds
Reset Game

You win!

Shots fired: 13
Total time: 9.9

You lose!

Shots fired: 13
Total time: 12.0

Fig. 6.2 | **Cannon Game** app `AlertDialog`s showing a win and a loss.

6.2 Test-Driving the Cannon Game App

Opening and Running the App
Open Eclipse and import the **Cannon Game** app project. Perform the following steps:

1. *Open the* **Import** *dialog.* Select **File > Import...** to open the **Import** dialog.
2. *Import the* **Cannon Game** *app's project.* In the **Import** dialog, expand the **General** node and select **Existing Projects into Workspace**, then click **Next >** to proceed to the **Import Projects** step. Ensure that **Select root directory** is selected, then click the **Browse...** button. In the **Browse for Folder** dialog, locate the `CannonGame` folder in the book's examples folder, select it and click **OK**. Click **Finish** to import the project into Eclipse. The project now appears in the **Package Explorer** window at the left side of the Eclipse window.
3. *Launch the* **Cannon Game** *app.* In Eclipse, right click the `CannonGame` project in the **Package Explorer** window, then select **Run As > Android Application** from the menu that appears.

Playing the Game
Tap the screen to aim and fire the cannon. You can fire a cannonball only if there is not another cannonball on the screen. If you're running this in an AVD, the mouse is your "finger." Try to destroy the target as fast as you can—the game ends if the timer runs out or you destroy all seven target pieces.

6.3 Technologies Overview

This section presents the new technologies that we use in the **Cannon Game** app in the order they're encountered in the chapter.

6.3.1 Attaching a Custom `View` to a Layout

You can create a *custom view* by extending class `View` or one of its subclasses, as we do with class `CannonView` (Section 6.8), which extends `SurfaceView` (discussed shortly). To add a custom component to a layout's XML file, you must provide its *fully qualified* name (i.e., its package and class name), so the custom `View`'s class must exist before you add it to the layout. We demonstrate how to create the `CannonView` class and add it to a layout in Section 6.4.3.

6.3.2 Using the Resource Folder `raw`

Media files, such as the sounds used in the **Cannon Game** app, are placed in the app's resource folder **`res/raw`**. Section 6.4.5 discusses how to create this folder. You'll then drag the app's sound files into it.

6.3.3 `Activity` and `Fragment` Lifecycle Methods

When a `Fragment` is attached to an `Activity` as we did in Chapter 5 and will do in this chapter, its lifecycle is tied to that of its parent `Activity`. There are six `Activity` lifecycle methods that have corresponding `Fragment` lifecycle methods—`onCreate`, `onStart`, `onResume`, `onPause`, `onStop` and `onDestroy`. When the system calls these methods on an `Activity`, it will also call these corresponding methods (and potentially other `Fragment` lifecycle methods) on all of the `Activity`'s attached `Fragments`.

This app uses `Fragment` lifecycle methods `onPause` and `onDestroy`. An `Activity`'s **`onPause`** method is called when *another* `Activity` receives the focus, which pauses the one that loses the focus and sends it to the background. When an `Activity` hosts `Fragments` and the `Activity` is paused, all of its `Fragments`' **`onPause`** methods are called. In this app, the `CannonView` is displayed in a `CannonGameFragment` (Section 6.7). We override `onPause` to suspend game play in the `CannonView` so that the game does not continue executing when the user cannot interact with it—this saves battery power. Many `Activity` lifecycle methods have corresponding methods in a `Fragment`'s lifecycle.

When an `Activity` is shut down, its **`onDestroy`** method is called, which in turn calls the **`onDestroy`** methods of all the `Fragments` hosted by the `Activity`. We use this method in the `CannonFragment` to *release* the `CannonView`'s sound resources.

We discuss other `Activity` and `Fragment` lifecycle methods as we need them. For more information on the complete `Activity` lifecycle, visit:

```
http://developer.android.com/reference/android/app/Activity.html
   #ActivityLifecycle
```

and for more information about the complete `Fragment` lifecycle, visit:

```
http://developer.android.com/guide/components/fragments.html
   #Lifecycle
```

6.3.4 Overriding `View` Method `onTouchEvent`

Users interact with this app by touching the device's screen. A *touch* aligns the cannon to face the touch point on the screen, then fires the cannon. To process simple touch events for the `CannonView`, you'll override `View` method **`onTouchEvent`** (Section 6.8.13), then use constants from class **`MotionEvent`** (package `android.view`) to test which type of event occurred and process it accordingly.

6.3.5 Adding Sound with `SoundPool` and `AudioManager`

An app's sound effects are managed with a **`SoundPool`** (package `android.media`), which can be used to *load*, *play* and *unload* sounds. Sounds are played using one of Android's audio streams for *alarms*, *music*, *notifications*, *phone rings*, *system sounds*, *phone calls* and more. The Android documentation recommends that games use the *music audio stream* to play sounds. We use the `Activity`'s **`setVolumeControlStream`** method to specify that the game's volume can be controlled with the device's volume keys. The method receives a constant from class **`AudioManager`** (package `android.media`), which provides access to the device's volume and phone ringer controls.

6.3.6 Frame-by-Frame Animation with Threads, `SurfaceView` and `SurfaceHolder`

This app *performs its animations manually* by updating the game elements from a separate thread of execuion. To do this, we use a subclass of `Thread` with a `run` method that directs our custom `CannonView` to update the positions of the game's elements, then draws them. The `run` method drives the *frame-by-frame animations*—this is known as the **game loop**.

Normally, all updates to an app's user interface must be performed in the GUI thread of execution. In Android, it's important to minimize the amount of work you do in the GUI thread to ensure that the GUI remains responsive and does not display ANR (Application Not Responding) dialogs. However, games often require complex logic that should be performed in separate threads of execution and those threads often need to draw to the screen. For such cases, Android provides class **`SurfaceView`**—a subclass of `View` to which a thread can draw, then indicate that the results should be displayed in the GUI thread. You manipulate a `SurfaceView` via an object of class **`SurfaceHolder`**, which enables you to obtain a `Canvas` on which you can draw graphics. Class `SurfaceHolder` also provides methods that give a thread *exclusive access* to the `Canvas` for drawing—only one thread at a time can draw to a `SurfaceView`. Each `SurfaceView` subclass should implement the interface **`SurfaceHolder.Callback`**, which contains methods that are called when the `SurfaceView` is *created*, *changed* (e.g., its size or orientation changes) or *destroyed*.

6.3.7 Simple Collision Detection

The `CannonView` performs simple *collision detection* to determine whether the cannonball has collided with any of the `CannonView`'s edges, with the blocker or with a section of the target. These techniques are presented in Section 6.8. Game-development frameworks typically provide more sophisticated "pixel-perfect" collision-detection capabilities. There are many open-source game-development frameworks available.

6.3.8 Drawing Graphics Using `Paint` and `Canvas`

We use methods of class **`Canvas`** (package `android.graphics`) to draw text, lines and circles. `Canvas` methods draw on a `View`'s **`Bitmap`**. Each drawing method in class `Canvas` uses an object of class **`Paint`** (package `android.graphics`) to specify drawing characteristics, including color, line thickness, font size and more. These capabilities are presented with the `drawGameElements` method in Section 6.8. For more details on the drawing characteristics you can specify with a `Paint` object, visit

```
    http://developer.android.com/reference/android/graphics/Paint.html
```

6.4 Building the App's GUI and Resource Files

In this section, you'll create the app's resource files and `main.xml` layout file.

6.4.1 Creating the Project

Begin by creating a new Android project named `CannonGame`. Specify the following values in the **New Android Project** dialog:

- **Application Name:** `Cannon Game`
- **Project Name:** `CannonGame`
- **Package Name:** `com.deitel.cannongame`
- **Minimum Required SDK:** `API18: Android 4.3`
- **Target SDK:** `API19: Android 4.4`
- **Compile With:** `API19: Android 4.4`
- **Theme:** `Holo Light with Dark Action Bar`

In the **New Android Project** dialog's second **New Android Application** step, leave the default settings, and press **Next >**. In the **Configure Launcher Icon** step, select an app icon image, then press **Next >**. In the **Create Activity** step, select **Blank Activity**, then press **Next >**. In the **Blank Activity** step, leave the default settings and click **Finish** to create the project. Open `activity_main.xml` in the **Graphical Layout** editor and select **Nexus 4** from the screen-type drop-down list. Once again, we'll use this device as the basis for our design.

Configure the App for Portrait Orientation

The cannon game is designed to work best in portrait orientation. Follow the steps you performed in Section 3.6 to set the app's screen orientation to portrait.

6.4.2 `strings.xml`

You created `String` resources in earlier chapters, so we show only a table (Fig. 6.3) of the `String` resource names and corresponding values here. Double click `strings.xml` in the `res/values` folder to display the resource editor for creating these `String` resources.

Resource name	Value
`results_format`	`Shots fired: %1$d\nTotal time: %2$.1f`
`reset_game`	`Reset Game`
`win`	`You win!`
`lose`	`You lose!`
`time_remaining_format`	`Time remaining: %.1f seconds`

Fig. 6.3 | `String` resources used in the **Cannon Game** app.

6.4.3 `fragment_game.xml`

The `fragment_game.xml` layout for the `CannonGameFragment` contains a `FrameLayout` that displays the `CannonView`. A **`FrameLayout`** is designed to display only one `View`—in this case, the `CannonView`. In this section, you'll create `CannonGameFragment`'s layout and the `CannonView` class. To add the `fragment_game.xml` layout, perform the following steps:

1. Expand the project's `res/layout` node in the **Package Explorer**.
2. Right click the `layout` folder and select **New > Android XML File** to display the **New Android XML File** dialog.
3. In the dialog's **File** field, enter `fragment_game.xml`
4. In the **Root Element** section, select **FrameLayout**, then click **Finish**.
5. From the **Palette**'s **Advanced** section, drag a **view** (with a lowercase **v**) onto the design area.
6. The previous step displays the **Choose Custom View Class** dialog. In that dialog, click **Create New...** to display the **New Java Class** dialog.
7. In the **Name** field, enter `CannonView`. In the **Superclass** field, change the superclass from `android.view.View` to `android.view.SurfaceView`. Ensure that **Constructors from superclass** is checked, then click **Finish**. This creates and opens `CannonView.java`. We'll be using only the two-argument constructor, so delete the other two. Save and close `CannonView.java`.

8. In fragment_game.xml, select **view1** in the **Outline** window. In the **Properties** window's **Layout Parameters** section, set **Width** and **Height** to match_parent.
9. In the **Outline** window, right click **view1**, select **Edit ID...**, rename view1 as cannonView and click **OK**.
10. Save fragment_game.xml.

6.4.4 activity_main.xml

The activity_main.xml layout for this app's MainActivity contains only the CannonGameFragment. To add this Fragment to the layout:

1. Open activity_main.xml in the **Graphical Layout** editor, then follow the steps in Section 2.5.2 to switch from a FrameLayout to a RelativeLayout.
2. From the **Palette**'s **Layouts** section, drag a **Fragment** onto the design area or onto the RelativeLayout node in the **Outline** window.
3. The preceding step displays the **Choose Fragment Class** dialog. Click **Create New...** to display the **New Java Class** dialog.
4. Enter CannonGameFragment in the dialog's **Name** field, change the **Superclass** field's value to android.app.Fragment and click **Finish** to create the class. The IDE opens the Java file for the class, which you can close for now.
5. Save activity_main.xml.

6.4.5 Adding the Sounds to the App

As we mentioned previously, sound files are stored in the app's res/raw folder. This app uses three sound files—blocker_hit.wav, target_hit.wav and cannon_fire.wav—which are located with the book's examples in the sounds folder. To add these files to your project:

1. Right click the app's res folder, then select **New > Folder**.
2. Specify the folder name raw and click **Finish** to create the folder.
3. Drag the sound files into the res/raw folder.

6.5 Class Line Maintains a Line's Endpoints

This app consists of four classes:

- Line (Fig. 6.4)
- MainActivity (the Activity subclass; Section 6.6)
- CannonGameFragment (Section 6.7), and
- CannonView (Section 6.8)

In this section, we discuss class Line, which represents a line's starting and ending Points. Objects of this class define the game's blocker and target. To add class Line to the project:

1. Expand the project's src node in the **Package Explorer**.
2. Right click the package (com.deitel.cannongame) and select **New > Class** to display the **New Java Class** dialog.
3. In the dialog's **Name** field, enter Line and click **Finish**.
4. Enter the code in Fig. 6.4 into the Line.java file. The default Point constructor sets a Point's public x and y instance variables to 0.

```java
// Line.java
// Class Line represents a line with two endpoints.
package com.deitel.cannongame;

import android.graphics.Point;

public class Line
{
   public Point start = new Point(); // start Point--(0,0) by default
   public Point end = new Point(); // end Point--(0,0) by default
} // end class Line
```

Fig. 6.4 | Class `Line` represents a line with two endpoints.

6.6 `MainActivity` Subclass of `Activity`

Class `MainActivity` (Fig. 6.5) is host for the **Cannon Game** app's `CannonGameFragment`. In this app, we override only `Activity` method `onCreate`, which inflates the GUI.

```java
// MainActivity.java
// MainActivity displays the CannonGameFragment
package com.deitel.cannongame;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity
{
   // called when the app first launches
   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState); // call super's onCreate method
      setContentView(R.layout.activity_main); // inflate the layout
   }
} // end class MainActivity
```

Fig. 6.5 | `MainActivity` displays the `CannonGameFragment`.

6.7 `CannonGameFragment` Subclass of `Fragment`

Class `CannonGameFragment` (Fig. 6.6) overrides four `Fragment` methods:

- `onCreateView` (lines 17–28)—As you learned in Section 5.3.3, this method is called after a `Fragment`'s `onCreate` method to build and return a `View` containing the `Fragment`'s GUI. Lines 22–23 inflate the GUI. Line 26 gets a reference to the `CannonGameFragment`'s `CannonView` so that we can call its methods.
- `onActivityCreated` (lines 31–38)—This method is called after the `Fragment`'s host `Activity` is created. Line 37 calls the `Activity`'s `setVolumeControlStream` method to allow the game's audio volume to be controlled by the device's volume keys.

- onPause (lines 41–46)—When the MainActivity is sent to the *background* (and thus, paused), the CannonGameFragment's method onPause executes. Line 45 calls the CannonView's stopGame method (Section 6.8.11) to stop the game loop.
- onDestroy (lines 49–54)—When the MainActivity is destroyed, its onDestroy method calls the CannonGameFragment's onDestroy. Line 46 calls the CannonView's releaseResources method (Section 6.8.11) to release the sound resources.

```
 1  // CannonGameFragment.java
 2  // CannonGameFragment creates and manages a CannonView
 3  package com.deitel.cannongame;
 4
 5  import android.app.Fragment;
 6  import android.media.AudioManager;
 7  import android.os.Bundle;
 8  import android.view.LayoutInflater;
 9  import android.view.View;
10  import android.view.ViewGroup;
11
12  public class CannonGameFragment extends Fragment
13  {
14     private CannonView cannonView; // custom view to display the game
15
16     // called when Fragment's view needs to be created
17     @Override
18     public View onCreateView(LayoutInflater inflater, ViewGroup container,
19        Bundle savedInstanceState)
20     {
21        super.onCreateView(inflater, container, savedInstanceState);
22        View view =
23           inflater.inflate(R.layout.fragment_game, container, false);
24
25        // get the CannonView
26        cannonView = (CannonView) view.findViewById(R.id.cannonView);
27        return view;
28     }
29
30     // set up volume control once Activity is created
31     @Override
32     public void onActivityCreated(Bundle savedInstanceState)
33     {
34        super.onActivityCreated(savedInstanceState);
35
36        // allow volume keys to set game volume
37        getActivity().setVolumeControlStream(AudioManager.STREAM_MUSIC);
38     }
39
40     // when MainActivity is paused, CannonGameFragment terminates the game
41     @Override
42     public void onPause()
43     {
```

Fig. 6.6 | CannonGameFragment creates and manages a CannonView. (Part 1 of 2.)

```
      super.onPause();
      cannonView.stopGame(); // terminates the game
   }

   // when MainActivity is paused, CannonGameFragment releases resources
   @Override
   public void onDestroy()
   {
      super.onDestroy();
      cannonView.releaseResources();
   }
} // end class CannonGameFragment
```

Fig. 6.6 | CannonGameFragment creates and manages a CannonView. (Part 2 of 2.)

6.8 CannonView Subclass of View

Class CannonView (Figs. 6.7–6.20) is a custom subclass of View that implements the **Cannon Game**'s logic and draws game objects on the screen.

6.8.1 package and import Statements

Figure 6.7 lists the package statement and the import statements for class CannonView. Section 6.3 discussed the key new classes and interfaces that class CannonView uses. We've highlighted them in Fig. 6.7.

```
// CannonView.java
// Displays and controls the Cannon Game
package com.deitel.cannongame;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.content.Context;
import android.content.DialogInterface;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Point;
import android.media.AudioManager;
import android.media.SoundPool;
import android.os.Bundle;
import android.util.AttributeSet;
import android.util.Log;
import android.util.SparseIntArray;
import android.view.MotionEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

```

Fig. 6.7 | CannonView class's package and import statements.

6.8.2 Instance Variables and Constants

Figure 6.8 lists the large number of class CannonView's constants and instance variables. Most are self documenting, but we'll explain each as we encounter it in the discussion.

```
public class CannonView extends SurfaceView
   implements SurfaceHolder.Callback
{
   private static final String TAG = "CannonView"; // for logging errors

   private CannonThread cannonThread; // controls the game loop
   private Activity activity; // to display Game Over dialog in GUI thread
   private boolean dialogIsDisplayed = false;

   // constants for game play
   public static final int TARGET_PIECES = 7; // sections in the target
   public static final int MISS_PENALTY = 2; // seconds deducted on a miss
   public static final int HIT_REWARD = 3; // seconds added on a hit

   // variables for the game loop and tracking statistics
   private boolean gameOver; // is the game over?
   private double timeLeft; // time remaining in seconds
   private int shotsFired; // shots the user has fired
   private double totalElapsedTime; // elapsed seconds

   // variables for the blocker and target
   private Line blocker; // start and end points of the blocker
   private int blockerDistance; // blocker distance from left
   private int blockerBeginning; // blocker top-edge distance from top
   private int blockerEnd; // blocker bottom-edge distance from top
   private int initialBlockerVelocity; // initial blocker speed multiplier
   private float blockerVelocity; // blocker speed multiplier during game

   private Line target; // start and end points of the target
   private int targetDistance; // target distance from left
   private int targetBeginning; // target distance from top
   private double pieceLength; // length of a target piece
   private int targetEnd; // target bottom's distance from top
   private int initialTargetVelocity; // initial target speed multiplier
   private float targetVelocity; // target speed multiplier

   private int lineWidth; // width of the target and blocker
   private boolean[] hitStates; // is each target piece hit?
   private int targetPiecesHit; // number of target pieces hit (out of 7)

   // variables for the cannon and cannonball
   private Point cannonball; // cannonball image's upper-left corner
   private int cannonballVelocityX; // cannonball's x velocity
   private int cannonballVelocityY; // cannonball's y velocity
   private boolean cannonballOnScreen; // whether cannonball on the screen
   private int cannonballRadius; // cannonball's radius
   private int cannonballSpeed; // cannonball's speed
   private int cannonBaseRadius; // cannon base's radius
```

Fig. 6.8 | CannonView class's fields. (Part 1 of 2.)

```
73       private int cannonLength; // cannon barrel's length
74       private Point barrelEnd; // the endpoint of the cannon's barrel
75       private int screenWidth;
76       private int screenHeight;
77
78       // constants and variables for managing sounds
79       private static final int TARGET_SOUND_ID = 0;
80       private static final int CANNON_SOUND_ID = 1;
81       private static final int BLOCKER_SOUND_ID = 2;
82       private SoundPool soundPool; // plays sound effects
83       private SparseIntArray soundMap; // maps IDs to SoundPool
84
85       // Paint variables used when drawing each item on the screen
86       private Paint textPaint; // Paint used to draw text
87       private Paint cannonballPaint; // Paint used to draw the cannonball
88       private Paint cannonPaint; // Paint used to draw the cannon
89       private Paint blockerPaint; // Paint used to draw the blocker
90       private Paint targetPaint; // Paint used to draw the target
91       private Paint backgroundPaint; // Paint used to clear the drawing area
92
```

Fig. 6.8 | CannonView class's fields. (Part 2 of 2.)

6.8.3 Constructor

Figure 6.9 shows class CannonView's constructor. When a View is inflated, its constructor is called with a Context and an AttributeSet as arguments. The Context is the Activity that displays the CannonGameFragment containing the CannonView, and the **AttributeSet** (package android.util) contains the CannonView attribute values that are set in the layout's XML document. These arguments are passed to the superclass constructor (line 96) to ensure that the custom View is properly configured with the values of any standard View attributes specified in the XML. Line 97 stores a reference to the MainActivity so we can use it at the end of a game to display an AlertDialog from the Activity's GUI thread.

```
93       // public constructor
94       public CannonView(Context context, AttributeSet attrs)
95       {
96          super(context, attrs); // call superclass constructor
97          activity = (Activity) context; // store reference to MainActivity
98
99          // register SurfaceHolder.Callback listener
100         getHolder().addCallback(this);
101
102         // initialize Lines and Point representing game items
103         blocker = new Line(); // create the blocker as a Line
104         target = new Line(); // create the target as a Line
105         cannonball = new Point(); // create the cannonball as a Point
106
107         // initialize hitStates as a boolean array
108         hitStates = new boolean[TARGET_PIECES];
```

Fig. 6.9 | CannonView constructor. (Part 1 of 2.)

```
109
110       // initialize SoundPool to play the app's three sound effects
111       soundPool = new SoundPool(1, AudioManager.STREAM_MUSIC, 0);
112
113       // create Map of sounds and pre-load sounds
114       soundMap = new SparseIntArray(3); // create new SparseIntArray
115       soundMap.put(TARGET_SOUND_ID,
116          soundPool.load(context, R.raw.target_hit, 1));
117       soundMap.put(CANNON_SOUND_ID,
118          soundPool.load(context, R.raw.cannon_fire, 1));
119       soundMap.put(BLOCKER_SOUND_ID,
120          soundPool.load(context, R.raw.blocker_hit, 1));
121
122       // construct Paints for drawing text, cannonball, cannon,
123       // blocker and target; these are configured in method onSizeChanged
124       textPaint = new Paint();
125       cannonPaint = new Paint();
126       cannonballPaint = new Paint();
127       blockerPaint = new Paint();
128       targetPaint = new Paint();
129       backgroundPaint = new Paint();
130    } // end CannonView constructor
131
```

Fig. 6.9 | CannonView constructor. (Part 2 of 2.)

Registering the **SurfaceHolder.Callback** *Listener*

Line 100 registers this (i.e., the CannonView) as the object that implements SurfaceHolder.Callback to receive the method calls that indicate when the SurfaceView is *created*, *updated* and *destroyed*. Inherited SurfaceView method **getHolder** returns the SurfaceHolder object for managing the SurfaceView, and SurfaceHolder method **addCallback** stores the object that implements interface SurfaceHolder.Callback.

Creating the **blocker**, **target** *and* **cannonball**

Lines 103–105 create the blocker and target as Lines and the cannonball as a Point. Next, we create boolean array hitStates to keep track of which of the target's seven pieces have been hit (and thus should not be drawn).

Configuring the **SoundPool** *and Loading the Sounds*

Lines 111–120 configure the sounds that we use in the app. First, we create the SoundPool that's used to load and play the app's sound effects. The constructor's first argument represents the maximum number of simultaneous sound streams that can play at once. We play only one sound at a time, so we pass 1. The second argument specifies which audio stream will be used to play the sounds. There are seven sound streams identified by constants in class AudioManager, but the documentation for class SoundPool recommends using the stream for playing music (AudioManager.STREAM_MUSIC) for sound in games. The last argument represents the sound quality, but the documentation indicates that this value is not currently used and 0 should be specified as the default value.

Line 114 creates a SparseIntArray (soundMap), which maps integer keys to integer values. SparseIntArray is similar to—but more efficient than—a HashMap<Integer, Integer> for small numbers of key–value pairs. In this case, we map the sound keys (defined in lines Fig. 6.8, 79–81) to the loaded sounds' IDs, which are represented by the return values of the `SoundPool`'s **`load` method** (called in Fig. 6.9, lines 116, 118 and 120). Each sound ID can be used to *play* a sound (and later to return its resources to the system). `SoundPool` method `load` receives three arguments—the application's `Context`, a resource ID representing the sound file to load and the sound's priority. According to the documentation for this method, the last argument is not currently used and should be specified as `1`.

Creating the `Paint` Objects Used to Draw Game Elements

Lines 124–129 create the `Paint` objects that are used when drawing the game's elements. We configure these in method `onSizeChanged` (Section 6.8.4), because some of the `Paint` settings depend on scaling the game elements based on the device's screen size.

6.8.4 Overriding `View` Method `onSizeChanged`

Figure 6.10 overrides class `View`'s **`onSizeChanged` method**, which is called whenever the `View`'s size changes, including when the `View` is first added to the `View` hierarchy as the layout is inflated. This app always displays in *portrait mode*, so `onSizeChanged` is called only once when the activity's `onCreate` method inflates the GUI. The method receives the `View`'s new width and height and its old width and height—when this method is called the first time, the old width and height are 0. The calculations performed here *scale* the game's on-screen elements based on the device's pixel width and height. We arrived at our scaling factors via trial and error, choosing values that made the game elements look nice on the screen. Lines 170–175 configure the `Paint` objects that are used to specify drawing characteristics for the game's elements. After the calculations, line 177 calls method `newGame` (Fig. 6.11).

```
132    // called by surfaceChanged when the size of the SurfaceView changes,
133    // such as when it's first added to the View hierarchy
134    @Override
135    protected void onSizeChanged(int w, int h, int oldw, int oldh)
136    {
137       super.onSizeChanged(w, h, oldw, oldh);
138
139       screenWidth = w; // store CannonView's width
140       screenHeight = h; // store CannonView's height
141       cannonBaseRadius = h / 18; // cannon base radius 1/18 screen height
142       cannonLength = w / 8; // cannon length 1/8 screen width
143
144       cannonballRadius = w / 36; // cannonball radius 1/36 screen width
145       cannonballSpeed = w * 3 / 2; // cannonball speed multiplier
146
147       lineWidth = w / 24; // target and blocker 1/24 screen width
148
149       // configure instance variables related to the blocker
150       blockerDistance = w * 5 / 8; // blocker 5/8 screen width from left
151       blockerBeginning = h / 8; // distance from top 1/8 screen height
152       blockerEnd = h * 3 / 8; // distance from top 3/8 screen height
153       initialBlockerVelocity = h / 2; // initial blocker speed multiplier
154       blocker.start = new Point(blockerDistance, blockerBeginning);
155       blocker.end = new Point(blockerDistance, blockerEnd);
```

Fig. 6.10 | Overridden `onSizeChanged` method. (Part 1 of 2.)

```
156
157       // configure instance variables related to the target
158       targetDistance = w * 7 / 8; // target 7/8 screen width from left
159       targetBeginning = h / 8; // distance from top 1/8 screen height
160       targetEnd = h * 7 / 8; // distance from top 7/8 screen height
161       pieceLength = (targetEnd - targetBeginning) / TARGET_PIECES;
162       initialTargetVelocity = -h / 4; // initial target speed multiplier
163       target.start = new Point(targetDistance, targetBeginning);
164       target.end = new Point(targetDistance, targetEnd);
165
166       // endpoint of the cannon's barrel initially points horizontally
167       barrelEnd = new Point(cannonLength, h / 2);
168
169       // configure Paint objects for drawing game elements
170       textPaint.setTextSize(w / 20); // text size 1/20 of screen width
171       textPaint.setAntiAlias(true); // smoothes the text
172       cannonPaint.setStrokeWidth(lineWidth * 1.5f); // set line thickness
173       blockerPaint.setStrokeWidth(lineWidth); // set line thickness
174       targetPaint.setStrokeWidth(lineWidth); // set line thickness
175       backgroundPaint.setColor(Color.WHITE); // set background color
176
177       newGame(); // set up and start a new game
178    } // end method onSizeChanged
179
```

Fig. 6.10 | Overridden onSizeChanged method. (Part 2 of 2.)

6.8.5 Method newGame

Method newGame (Fig. 6.11) resets the instance variables that are used to control the game. If variable gameOver is true, which occurs only *after* the first game completes, line 203 resets gameOver and lines 204–205 create a new CannonThread and start it to begin the *game loop* that controls the game. You'll learn more about this in Section 6.8.14.

```
180    // reset all the screen elements and start a new game
181    public void newGame()
182    {
183       // set every element of hitStates to false--restores target pieces
184       for (int i = 0; i < TARGET_PIECES; i++)
185          hitStates[i] = false;
186
187       targetPiecesHit = 0; // no target pieces have been hit
188       blockerVelocity = initialBlockerVelocity; // set initial velocity
189       targetVelocity = initialTargetVelocity; // set initial velocity
190       timeLeft = 10; // start the countdown at 10 seconds
191       cannonballOnScreen = false; // the cannonball is not on the screen
192       shotsFired = 0; // set the initial number of shots fired
193       totalElapsedTime = 0.0; // set the time elapsed to zero
194
195       // set the start and end Points of the blocker and target
196       blocker.start.set(blockerDistance, blockerBeginning);
197       blocker.end.set(blockerDistance, blockerEnd);
```

Fig. 6.11 | CannonView method newGame. (Part 1 of 2.)

```
      target.start.set(targetDistance, targetBeginning);
      target.end.set(targetDistance, targetEnd);

      if (gameOver) // starting a new game after the last game ended
      {
         gameOver = false; // the game is not over
         cannonThread = new CannonThread(getHolder()); // create thread
         cannonThread.start(); // start the game loop thread
      } // end if
   } // end method newGame

```

Fig. 6.11 | `CannonView` method `newGame`. (Part 2 of 2.)

6.8.6 Method `updatePositions`

Method `updatePositions` (Fig. 6.12) is called by the `CannonThread`'s `run` method (Section 6.8.14) to update the on-screen elements' positions and to perform simple *collision detection*. The new locations of the game elements are calculated based on the elapsed time in milliseconds between the previous and current animation frames. This enables the game to update the amount by which each game element moves based on the device's *refresh rate*. We discuss this in more detail when we cover game loops in Section 6.8.14.

```
   // called repeatedly by the CannonThread to update game elements
   private void updatePositions(double elapsedTimeMS)
   {
      double interval = elapsedTimeMS / 1000.0; // convert to seconds

      if (cannonballOnScreen) // if there is currently a shot fired
      {
         // update cannonball position
         cannonball.x += interval * cannonballVelocityX;
         cannonball.y += interval * cannonballVelocityY;

         // check for collision with blocker
         if (cannonball.x + cannonballRadius > blockerDistance &&
            cannonball.x - cannonballRadius < blockerDistance &&
            cannonball.y + cannonballRadius > blocker.start.y &&
            cannonball.y - cannonballRadius < blocker.end.y)
         {
            cannonballVelocityX *= -1; // reverse cannonball's direction
            timeLeft -= MISS_PENALTY; // penalize the user

            // play blocker sound
            soundPool.play(soundMap.get(BLOCKER_SOUND_ID), 1, 1, 1, 0, 1f);
         }
         // check for collisions with left and right walls
         else if (cannonball.x + cannonballRadius > screenWidth ||
            cannonball.x - cannonballRadius < 0)
         {
```

Fig. 6.12 | `CannonView` method `updatePositions`. (Part 1 of 3.)

```
            cannonballOnScreen = false; // remove cannonball from screen
         }
         // check for collisions with top and bottom walls
         else if (cannonball.y + cannonballRadius > screenHeight ||
            cannonball.y - cannonballRadius < 0)
         {
            cannonballOnScreen = false; // remove cannonball from screen
         }
         // check for cannonball collision with target
         else if (cannonball.x + cannonballRadius > targetDistance &&
            cannonball.x - cannonballRadius < targetDistance &&
            cannonball.y + cannonballRadius > target.start.y &&
            cannonball.y - cannonballRadius < target.end.y)
         {
            // determine target section number (0 is the top)
            int section =
               (int) ((cannonball.y - target.start.y) / pieceLength);

            // check if the piece hasn't been hit yet
            if ((section >= 0 && section < TARGET_PIECES) &&
               !hitStates[section])
            {
               hitStates[section] = true; // section was hit
               cannonballOnScreen = false; // remove cannonball
               timeLeft += HIT_REWARD; // add reward to remaining time

               // play target hit sound
               soundPool.play(soundMap.get(TARGET_SOUND_ID), 1,
                  1, 1, 0, 1f);

               // if all pieces have been hit
               if (++targetPiecesHit == TARGET_PIECES)
               {
                  cannonThread.setRunning(false); // terminate thread
                  showGameOverDialog(R.string.win); // show winning dialog
                  gameOver = true;
               }
            }
         }
      }

      // update the blocker's position
      double blockerUpdate = interval * blockerVelocity;
      blocker.start.y += blockerUpdate;
      blocker.end.y += blockerUpdate;

      // update the target's position
      double targetUpdate = interval * targetVelocity;
      target.start.y += targetUpdate;
      target.end.y += targetUpdate;

```

Fig. 6.12 | CannonView method updatePositions. (Part 2 of 3.)

```
287       // if the blocker hit the top or bottom, reverse direction
288       if (blocker.start.y < 0 || blocker.end.y > screenHeight)
289          blockerVelocity *= -1;
290
291       // if the target hit the top or bottom, reverse direction
292       if (target.start.y < 0 || target.end.y > screenHeight)
293          targetVelocity *= -1;
294
295       timeLeft -= interval; // subtract from time left
296
297       // if the timer reached zero
298       if (timeLeft <= 0.0)
299       {
300          timeLeft = 0.0;
301          gameOver = true; // the game is over
302          cannonThread.setRunning(false); // terminate thread
303          showGameOverDialog(R.string.lose); // show the losing dialog
304       }
305    } // end method updatePositions
306
```

Fig. 6.12 | `CannonView` method `updatePositions`. (Part 3 of 3.)

Elapsed Time Since the Last Animation Frame

Line 212 converts the elapsed time since the last animation frame from milliseconds to seconds. This value is used to modify the positions of various game elements.

Checking for Collisions with the Blocker

Line 214 checks whether the cannonball is on the screen. If it is, we update its position by adding the distance it should have traveled since the last timer event. This is calculated by multiplying its velocity by the amount of time that passed (lines 217–218). Lines 221–224 check whether the cannonball has *collided* with the blocker. We perform simple *collision detection*, based on the rectangular boundary of the cannonball. There are four conditions that must be met if the cannonball is in contact with the blocker:

- The cannonball's *x*-coordinate plus the cannonball's radius must be greater than the blocker's distance from the left edge of the screen (`blockerDistance`) (line 221). This means that the cannonball has reached the blocker's distance from the left edge of the screen.
- The cannonball's *x*-coordinate minus the cannonball's radius must also be less than the blocker's distance from the left edge of the screen (line 222). This ensures that the cannonball has not yet passed the blocker.
- Part of the cannonball must be lower than the top of the blocker (line 223).
- Part of the cannonball must be higher than the bottom of the blocker (line 224).

If all these conditions are met, we *reverse* the cannonball's direction on the screen (line 226), *penalize* the user by *subtracting* `MISS_PENALTY` from `timeLeft`, then call `soundPool`'s **`play` method** to play the blocker hit sound—`BLOCKER_SOUND_ID` is used as the `soundMap` key to locate the sound's ID in the `SoundPool`.

Checking Whether the Cannonball Left the Screen
We remove the cannonball if it reaches any of the screen's edges. Lines 233–237 test whether the cannonball has *collided* with the left or right wall and, if it has, remove the cannonball from the screen. Lines 239–243 remove the cannonball if it collides with the top or bottom of the screen.

Checking for Collisions with the Target
We then check whether the cannonball has hit the target (lines 245–248). These conditions are similar to those used to determine whether the cannonball collided with the blocker. If the cannonball hit the target, lines 251–252 determine which *section* has been hit—dividing the distance between the cannonball and the bottom of the target by the length of a piece. This expression evaluates to 0 for the topmost section and 6 for the bottommost. We check whether that section was previously hit, using the hitStates array (line 256). If it wasn't, we set the corresponding hitStates element to true and remove the cannonball from the screen. We then add HIT_REWARD to timeLeft, increasing the game's time remaining, and play the target hit sound (TARGET_SOUND_ID). We increment targetPiecesHit, then determine whether it's equal to TARGET_PIECES (line 267). If so, the game is over, so we terminate the CannonThread by calling its setRunning method with the argument false, invoke method showGameOverDialog with the String resource ID representing the winning message and set gameOver to true.

Updating the Blocker and Target Positions
Now that all possible cannonball collisions have been checked, the blocker and target positions must be updated. Lines 278–280 change the blocker's position by multiplying blockerVelocity by the amount of time that has passed since the last update, and adding that value to the current *x*- and *y*-coordinates. Lines 283–285 do the same for the target. If the blocker has collided with the top or bottom wall, its direction is *reversed* by multiplying its velocity by -1 (lines 288–289). Lines 292–293 perform the same check and adjustment for the full length of the target, including any sections that have already been destroyed.

Updating the Time Left and Determining Whether Time Ran Out
We decrease timeLeft by the time that has passed since the prior animation frame (line 295). If timeLeft has reached zero, the game is over—we set timeLeft to 0.0 just in case it was negative; otherwise, sometimes a negative final time would display on the screen). Then we set gameOver to true, terminate the CannonThread by calling its setRunning method with the argument false and call method showGameOverDialog with the String resource ID representing the losing message.

6.8.7 Method fireCannonball

When the user *touches* the screen, method onTouchEvent (Section 6.8.13) calls fireCannonball (Fig. 6.13). If there's already a cannonball on the screen, the method *returns immediately*. Line 313 calls alignCannon to aim the cannon at the *touch point* and get the cannon's angle. Lines 316–317 "load the cannon" (that is, position the cannonball inside the cannon). Then, lines 320 and 323 calculate the horizontal and vertical components of the cannonball's velocity. Next, we set cannonballOnScreen to true so that the cannonball will be drawn by method drawGameElements (Fig. 6.15) and increment shotsFired. Finally, we play the cannon's firing sound (represented by the CANNON_SOUND_ID).

```
307    // fires a cannonball
308    public void fireCannonball(MotionEvent event)
309    {
310       if (cannonballOnScreen) // if a cannonball is already on the screen
311          return; // do nothing
312
313       double angle = alignCannon(event); // get the cannon barrel's angle
314
315       // move the cannonball to be inside the cannon
316       cannonball.x = cannonballRadius; // align x-coordinate with cannon
317       cannonball.y = screenHeight / 2; // centers ball vertically
318
319       // get the x component of the total velocity
320       cannonballVelocityX = (int) (cannonballSpeed * Math.sin(angle));
321
322       // get the y component of the total velocity
323       cannonballVelocityY = (int) (-cannonballSpeed * Math.cos(angle));
324       cannonballOnScreen = true; // the cannonball is on the screen
325       ++shotsFired; // increment shotsFired
326
327       // play cannon fired sound
328       soundPool.play(soundMap.get(CANNON_SOUND_ID), 1, 1, 1, 0, 1f);
329    } // end method fireCannonball
330
```

Fig. 6.13 | `CannonView` method `fireCannonball`.

6.8.8 Method `alignCannon`

Method `alignCannon` (Fig. 6.14) aims the cannon at the point where the user touched the screen. Line 335 gets the *x*- and *y*-coordinates of the *touch* from the `MotionEvent` argument. We compute the vertical distance of the touch from the center of the screen. If this is not zero, we calculate cannon barrel's `angle` from the horizontal (line 345). If the touch is on the lower-half of the screen we adjust the `angle` by `Math.PI` (line 349). We then use the `cannonLength` and the `angle` to determine the *x*- and *y*-coordinate values for the endpoint of the cannon's barrel—this is used to draw a line from the cannon base's center at the left edge of the screen to the cannon's barrel endpoint.

```
331    // aligns the cannon in response to a user touch
332    public double alignCannon(MotionEvent event)
333    {
334       // get the location of the touch in this view
335       Point touchPoint = new Point((int) event.getX(), (int) event.getY());
336
337       // compute the touch's distance from center of the screen
338       // on the y-axis
339       double centerMinusY = (screenHeight / 2 - touchPoint.y);
340
341       double angle = 0; // initialize angle to 0
342
```

Fig. 6.14 | `CannonView` method `alignCannon`. (Part 1 of 2.)

```
      // calculate the angle the barrel makes with the horizontal
      if (centerMinusY != 0) // prevent division by 0
         angle = Math.atan((double) touchPoint.x / centerMinusY);

      // if the touch is on the lower half of the screen
      if (touchPoint.y > screenHeight / 2)
         angle += Math.PI; // adjust the angle

      // calculate the endpoint of the cannon barrel
      barrelEnd.x = (int) (cannonLength * Math.sin(angle));
      barrelEnd.y =
         (int) (-cannonLength * Math.cos(angle) + screenHeight / 2);

      return angle; // return the computed angle
   } // end method alignCannon

```

Fig. 6.14 | CannonView method alignCannon. (Part 2 of 2.)

6.8.9 Method drawGameElements

The method drawGameElements (Fig. 6.15) draws the *cannon*, *cannonball*, *blocker* and *target* on the SurfaceView using the Canvas that the CannonThread (Section 6.8.14) obtains from the SurfaceView's SurfaceHolder.

```
   // draws the game to the given Canvas
   public void drawGameElements(Canvas canvas)
   {
      // clear the background
      canvas.drawRect(0, 0, canvas.getWidth(), canvas.getHeight(),
         backgroundPaint);

      // display time remaining
      canvas.drawText(getResources().getString(
         R.string.time_remaining_format, timeLeft), 30, 50, textPaint);

      // if a cannonball is currently on the screen, draw it
      if (cannonballOnScreen)
         canvas.drawCircle(cannonball.x, cannonball.y, cannonballRadius,
            cannonballPaint);

      // draw the cannon barrel
      canvas.drawLine(0, screenHeight / 2, barrelEnd.x, barrelEnd.y,
         cannonPaint);

      // draw the cannon base
      canvas.drawCircle(0, (int) screenHeight / 2,
         (int) cannonBaseRadius, cannonPaint);

      // draw the blocker
      canvas.drawLine(blocker.start.x, blocker.start.y, blocker.end.x,
         blocker.end.y, blockerPaint);
```

Fig. 6.15 | CannonView method drawGameElements. (Part 1 of 2.)

```
386
387       Point currentPoint = new Point(); // start of current target section
388
389       // initialize currentPoint to the starting point of the target
390       currentPoint.x = target.start.x;
391       currentPoint.y = target.start.y;
392
393       // draw the target
394       for (int i = 0; i < TARGET_PIECES; i++)
395       {
396          // if this target piece is not hit, draw it
397          if (!hitStates[i])
398          {
399             // alternate coloring the pieces
400             if (i % 2 != 0)
401                targetPaint.setColor(Color.BLUE);
402             else
403                targetPaint.setColor(Color.YELLOW);
404
405             canvas.drawLine(currentPoint.x, currentPoint.y, target.end.x,
406                (int) (currentPoint.y + pieceLength), targetPaint);
407          }
408
409          // move currentPoint to the start of the next piece
410          currentPoint.y += pieceLength;
411       }
412    } // end method drawGameElements
413
```

Fig. 6.15 | `CannonView` method `drawGameElements`. (Part 2 of 2.)

Clearing the **`Canvas`** *with Method* **`drawRect`**

First, we call `Canvas`'s **`drawRect` method** (lines 363–364) to clear the `Canvas` so that all the game elements can be displayed in their new positions. The method receives as arguments the rectangle's upper-left *x-y* coordinates, the rectangle's width and height, and the `Paint` object that specifies the drawing characteristics—recall that `backgroundPaint` sets the drawing color to white.

Displaying the Time Remaining with **`Canvas`** *Method* **`drawText`**

Next, we call `Canvas`'s **`drawText` method** (lines 367–368) to display the time remaining in the game. We pass as arguments the `String` to be displayed, the *x*- and *y*-coordinates at which to display it and the `textPaint` (configured in lines 170–171) to describe how the text should be rendered (that is, the text's font size, color and other attributes).

Drawing the Cannonball with **`Canvas`** *Method* **`drawCircle`**

If the cannonball is on the screen, lines 372–373 use `Canvas`'s **`drawCircle` method** to draw the cannonball in its current position. The first two arguments represent the coordinates of the circle's *center*. The third argument is the circle's *radius*. The last argument is the `Paint` object specifying the circle's drawing characteristics.

Drawing the Cannon Barrel, Blocker and Target with **Canvas** *Method* **drawLine**
We use Canvas's **drawLine method** to display the cannon *barrel* (lines 376–377), the *blocker* (lines 384–385) and the *target pieces* (lines 405–406). This method receives five parameters—the first four represent the *x-y* coordinates of the line's start and end, and the last is the Paint object specifying the line's characteristics, such as its thickness.

Drawing the Cannon Base with **Canvas** *Method* **drawCircle**
Lines 380–381 use Canvas's drawCircle method to draw the cannon's half-circle base by drawing a circle that's centered at the left edge of the screen—because a circle is displayed based on its center point, half of this circle is drawn off the left side of the SurfaceView.

Drawing the Target Sections with **Canvas** *Method* **drawLine**
Lines 390–411 draw the target sections. We iterate through the sections, drawing each in the correct color—blue for the odd-numbered pieces and yellow for the others. Only those sections that haven't been hit are displayed.

6.8.10 Method showGameOverDialog

When the game ends, the showGameOverDialog method (Fig. 6.16) displays a DialogFragment (using the techniques you learned in Section 5.6.9) containing an AlertDialog that indicates whether the player won or lost, the number of shots fired and the total time elapsed. The call to method setPositiveButton (lines 433–444) creates a reset button for starting a new game.

```
414    // display an AlertDialog when the game ends
415    private void showGameOverDialog(final int messageId)
416    {
417       // DialogFragment to display quiz stats and start new quiz
418       final DialogFragment gameResult =
419          new DialogFragment()
420          {
421             // create an AlertDialog and return it
422             @Override
423             public Dialog onCreateDialog(Bundle bundle)
424             {
425                // create dialog displaying String resource for messageId
426                AlertDialog.Builder builder =
427                   new AlertDialog.Builder(getActivity());
428                builder.setTitle(getResources().getString(messageId));
429
430                // display number of shots fired and total time elapsed
431                builder.setMessage(getResources().getString(
432                   R.string.results_format, shotsFired, totalElapsedTime));
433                builder.setPositiveButton(R.string.reset_game,
434                   new DialogInterface.OnClickListener()
435                   {
436                      // called when "Reset Game" Button is pressed
437                      @Override
438                      public void onClick(DialogInterface dialog, int which)
439                      {
```

Fig. 6.16 | CannonView method showGameOverDialog. (Part 1 of 2.)

```
440                         dialogIsDisplayed = false;
441                         newGame(); // set up and start a new game
442                      }
443                   } // end anonymous inner class
444                ); // end call to setPositiveButton
445
446                return builder.create(); // return the AlertDialog
447             } // end method onCreateDialog
448          }; // end DialogFragment anonymous inner class
449
450       // in GUI thread, use FragmentManager to display the DialogFragment
451       activity.runOnUiThread(
452          new Runnable() {
453             public void run()
454             {
455                dialogIsDisplayed = true;
456                gameResult.setCancelable(false); // modal dialog
457                gameResult.show(activity.getFragmentManager(), "results");
458             }
459          } // end Runnable
460       ); // end call to runOnUiThread
461    } // end method showGameOverDialog
462
```

Fig. 6.16 | `CannonView` method `showGameOverDialog`. (Part 2 of 2.)

The `onClick` method of the button's listener indicates that the dialog is no longer displayed and calls `newGame` to set up and start a new game. A dialog must be displayed from the GUI thread, so lines 451–460 call `Activity` method **`runOnUiThread`** to specify a `Runnable` that should execute in the GUI thread as soon as possible. The argument is an object of an anonymous inner class that implements `Runnable`. The `Runnable`'s `run` method indicates that the dialog is displayed and then displays it.

6.8.11 Methods `stopGame` and `releaseResources`

Class `CannonGameFragment`'s `onPause` and `onDestroy` methods (Section 6.7) call class `CannonView`'s `stopGame` and `releaseResources` methods (Fig. 6.17), respectively. Method `stopGame` (lines 464–468) is called from the main `Activity` to stop the game when the `Activity`'s `onPause` method is called—for simplicity, we don't store the game's state in this example. Method `releaseResources` (lines 471–475) calls the `SoundPool`'s **`release` method** to release the resources associated with the `SoundPool`.

```
463    // stops the game; called by CannonGameFragment's onPause method
464    public void stopGame()
465    {
466       if (cannonThread != null)
467          cannonThread.setRunning(false); // tell thread to terminate
468    }
469
```

Fig. 6.17 | `CannonView` methods `stopGame` and `releaseResources`. (Part 1 of 2.)

```
470     // releases resources; called by CannonGame's onDestroy method
471     public void releaseResources()
472     {
473        soundPool.release(); // release all resources used by the SoundPool
474        soundPool = null;
475     }
476
```

Fig. 6.17 | `CannonView` methods `stopGame` and `releaseResources`. (Part 2 of 2.)

6.8.12 Implementing the `SurfaceHolder.Callback` Methods

Figure 6.18 implements the **`surfaceChanged`**, **`surfaceCreated`** and **`surfaceDestroyed`** methods of interface `SurfaceHolder.Callback`. Method `surfaceChanged` has an empty body in this app because the app is *always* displayed in *portrait orientation*. This method is called when the `SurfaceView`'s size or orientation changes, and would typically be used to redisplay graphics based on those changes. Method `surfaceCreated` (lines 485–494) is called when the `SurfaceView` is created—e.g., when the app first loads or when it resumes from the background. We use `surfaceCreated` to create and start the `CannonThread` to begin the game loop. Method `surfaceDestroyed` (lines 497–515) is called when the `SurfaceView` is destroyed—e.g., when the app terminates. We use the method to ensure that the `CannonThread` terminates properly. First, line 502 calls `CannonThread`'s `setRunning` method with `false` as an argument to indicate that the thread should *stop*, then lines 504–515 wait for the thead to *terminate*. This ensures that no attempt is made to draw to the `SurfaceView` once `surfaceDestroyed` completes execution.

```
477     // called when surface changes size
478     @Override
479     public void surfaceChanged(SurfaceHolder holder, int format,
480        int width, int height)
481     {
482     }
483
484     // called when surface is first created
485     @Override
486     public void surfaceCreated(SurfaceHolder holder)
487     {
488        if (!dialogIsDisplayed)
489        {
490           cannonThread = new CannonThread(holder); // create thread
491           cannonThread.setRunning(true); // start game running
492           cannonThread.start(); // start the game loop thread
493        }
494     }
495
496     // called when the surface is destroyed
497     @Override
498     public void surfaceDestroyed(SurfaceHolder holder)
499     {
```

Fig. 6.18 | Implementing the `SurfaceHolder.Callback` methods. (Part 1 of 2.)

```
      // ensure that thread terminates properly
      boolean retry = true;
      cannonThread.setRunning(false); // terminate cannonThread

      while (retry)
      {
         try
         {
            cannonThread.join(); // wait for cannonThread to finish
            retry = false;
         }
         catch (InterruptedException e)
         {
            Log.e(TAG, "Thread interrupted", e);
         }
      }
   } // end method surfaceDestroyed

```

Fig. 6.18 | Implementing the `SurfaceHolder.Callback` methods. (Part 2 of 2.)

6.8.13 Overriding View Method onTouchEvent

In this example, we override `View` method `onTouchEvent` (Fig. 6.19) to determine when the user touches the screen. The `MotionEvent` parameter contains information about the event that occurred. Line 523 uses the `MotionEvent`'s `getAction` method to determine which type of touch event occurred. Then, lines 526–527 determine whether the user touched the screen (`MotionEvent.ACTION_DOWN`) or dragged a finger across the screen (`MotionEvent.ACTION_MOVE`). In either case, line 529 calls the `cannonView`'s `fireCannonball` method to aim and fire the cannon toward that touch point. Line 532 then returns `true` to indicate that the touch event was handled.

```
518    // called when the user touches the screen in this Activity
519    @Override
520    public boolean onTouchEvent(MotionEvent e)
521    {
522       // get int representing the type of action which caused this event
523       int action = e.getAction();
524
525       // the user user touched the screen or dragged along the screen
526       if (action == MotionEvent.ACTION_DOWN ||
527          action == MotionEvent.ACTION_MOVE)
528       {
529          fireCannonball(e); // fire the cannonball toward the touch point
530       }
531
532       return true;
533    } // end method onTouchEvent
534
```

Fig. 6.19 | Overriding `View` method `onTouchEvent`. .

6.8.14 CannonThread: Using a Thread to Create a Game Loop

Figure 6.20 defines a subclass of `Thread` which updates the game. The thread maintains a reference to the `SurfaceView`'s `SurfaceHolder` (line 538) and a `boolean` indicating whether the thread is *running*. The class's `run` method (lines 556–587) drives the *frame-by-frame animations*—this is known as the *game loop*. Each update of the game elements on the screen is performed based on the number of milliseconds that have passed since the last update. Line 559 gets the system's current time in milliseconds when the thread begins running. Lines 561–586 loop until `threadIsRunning` is `false`.

```
535    // Thread subclass to control the game loop
536    private class CannonThread extends Thread
537    {
538       private SurfaceHolder surfaceHolder; // for manipulating canvas
539       private boolean threadIsRunning = true; // running by default
540
541       // initializes the surface holder
542       public CannonThread(SurfaceHolder holder)
543       {
544          surfaceHolder = holder;
545          setName("CannonThread");
546       }
547
548       // changes running state
549       public void setRunning(boolean running)
550       {
551          threadIsRunning = running;
552       }
553
554       // controls the game loop
555       @Override
556       public void run()
557       {
558          Canvas canvas = null; // used for drawing
559          long previousFrameTime = System.currentTimeMillis();
560
561          while (threadIsRunning)
562          {
563             try
564             {
565                // get Canvas for exclusive drawing from this thread
566                canvas = surfaceHolder.lockCanvas(null);
567
568                // lock the surfaceHolder for drawing
569                synchronized(surfaceHolder)
570                {
571                   long currentTime = System.currentTimeMillis();
572                   double elapsedTimeMS = currentTime - previousFrameTime;
573                   totalElapsedTime += elapsedTimeMS / 1000.0;
574                   updatePositions(elapsedTimeMS); // update game state
575                   drawGameElements(canvas); // draw using the canvas
```

Fig. 6.20 | `Runnable` that updates the game every `TIME_INTERVAL` milliseconds. (Part 1 of 2.)

```
576                    previousFrameTime = currentTime; // update previous time
577                 }
578              }
579              finally
580              {
581                 // display canvas's contents on the CannonView
582                 // and enable other threads to use the Canvas
583                 if (canvas != null)
584                    surfaceHolder.unlockCanvasAndPost(canvas);
585              }
586           } // end while
587        } // end method run
588     } // end nested class CannonThread
589  } // end class CannonView
```

Fig. 6.20 | `Runnable` that updates the game every `TIME_INTERVAL` milliseconds. (Part 2 of 2.)

First we obtain the `Canvas` for drawing on the `SurfaceView` by calling `SurfaceHolder` method **`lockCanvas`** (line 566). Only one thread at a time can draw to a `SurfaceView`. To ensure this, you must first *lock* the `SurfaceHolder` by specifying it as the expression in the parentheses of a `synchronized` block (line 569). Next, we get the current time in milliseconds, then calculate the elapsed time and add that to the total time so far—this will be used to help display the amount of time left in the game. Line 574 calls method `updatePositions` to move all the game elements, passing the elapsed time in milliseconds as an argument. This ensures that the game operates at the same speed *regardless of how fast the device is*. If the time between frames is larger (i.e, the device is slower), the game elements will move further when each frame of the animation is displayed. If the time between frames is smaller (i.e, the device is faster), the game elements will move less when each frame of the animation is displayed. Finally, line 575 draws the game elements using the `SurfaceView`'s `Canvas` and line 576 stores the `currentTime` as the `previousFrameTime` to prepare to calculate the elapsed time between this animation frame and the *next*.

6.9 Wrap-Up

In this chapter, you created the **Cannon Game** app, which challenges the player to destroy a seven-piece target before a 10-second time limit expires. The user aims and fires the cannon by touching the screen. To draw on the screen from a separate thread, you created a custom view by extending class `SurfaceView`. You learned that custom component class names must be fully qualified in the XML layout element that represents the component. We presented additional `Fragment` lifecycle methods. You learned that method `onPause` is called when a `Fragment` is paused and method `onDestroy` is called when the `Fragment` is destroyed. You handled touches by overriding `View`'s `onTouchEvent` method. You added sound effects to the app's `res/raw` folder and managed them with a `SoundPool`. You also used the system's `AudioManager` service to obtain the device's current music volume and use it as the playback volume.

This app manually performs its animations by updating the game elements on a `SurfaceView` from a separate thread of execution. To do this, you extended class `Thread` and created a `run` method that displays graphics by calling methods of class `Canvas`. You used the `SurfaceView`'s `SurfaceHolder` to obtain the appropriate `Canvas`. You also learned how to build a game loop that controls a game based on the amount of time that has elapsed between animation frames, so that the game will operate at the same overall speed on all devices, regardless of their processor speeds.

In Chapter 7, we present the **Doodlz** app, which uses Android's graphics capabilities to turn a device's screen into a *virtual canvas*. You'll also learn about Android 4.4's new immersive mode and printing capabilities.

Self-Review Exercises

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

a) An `Activity`'s __________ method is called when it is about to be killed.

b) To process simple touch events for an `Activity`, you can override class `Activity`'s `onTouchEvent` method then use constants from class __________ (package `android.view`) to test which type of event occurred and process it accordingly.

c) Each `SurfaceView` subclass should implement the interface __________, which contains methods that are called when the `SurfaceView` is created, changed (e.g., its size or orientation changes) or destroyed.

d) Each drawing method in class `Canvas` uses an object of class __________ to specify drawing characteristics like color and line thickness.

e) An Android app's sound effects are managed with a __________ class.

6.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The Android documentation recommends that games use the music audio stream to play sounds.

b) In Android, it's important to maximize the amount of work you do in the GUI thread to ensure that the GUI remains responsive and does not display ANR (Application Not Responding) dialogs.

c) A `Canvas` draws on a `View`'s `Bitmap`.

d) Format `String`s that contain multiple format specifiers must number the format specifiers for localization purposes.

e) There are seven sound streams identified by constants in class `AudioManager`, but the documentation for class `SoundPool` recommends using the stream for playing music (`AudioManager.STREAM_MUSIC`) for sound in games.

f) Custom component class names must be fully qualified in the XML layout element that represents the component.

Answers to Self-Review Exercises

6.1 a) `onDestroy`. b) `MotionEvent`. c) `SurfaceHolder.Callback`. d) `Paint`. e) `SoundPool`.

6.2 a) True. b) False. In Android, it's important to *minimize* the amount of work you do in the GUI thread to ensure that the GUI remains responsive and does not display ANR (Application Not Responding) dialogs. c) True. d) True. e) True. f) True.

Exercises

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

a) An application's sound effects are managed with a __________ of package `android.media`, which can be used to load, play and unload sounds.

b) We use methods of class __________ to draw text, lines and circles.

c) A(n) __________ allows an app to react to more sophisticated user interactions such as flings, double-taps, long presses and scrolls.

d) `Activity`'s __________ method specifies that an app's volume can be controlled with the device's volume keys and should be the same as the device's music playback volume. The method receives a constant from class `AudioManager` (package `android.media`).

e) Games often require complex logic that should be performed in separate threads of execution and those threads often need to draw to the screen. For such cases, Android provides class __________—a subclass of `View` to which any thread can draw.

f) Media files, such as the sounds used in an app, are placed in the resource folder __________.

6.4 State whether each of the following is *true* or *false*. If *false*, explain why.

a) When the system calls lifecycle methods on an `Activity`, it will also call the corresponding lifecycle methods on all of the `Activity`'s attached `Fragments`.

b) A `MotionEvent.ACTION_TOUCH` indicates that the user touched the screen and indicates that the user moved a finger across the screen (`MotionEvent.ACTION_MOVE`).

c) When a `View` is inflated, its constructor is called and passed a `Context` and an `AttributeSet` as arguments.

d) To add a custom component to a layout's XML file, you need not provide its fully qualified name (i.e., its package and class name).

e) When a game loop controls a game based on the amount of time that has elapsed between animation frames, the game will operate at different speeds as appropriate for each device.

6.5 *(Enhanced* **Cannon Game** *App)* Modify the **Cannon Game** app as follows:

a) Use images for the cannon base and cannonball.

b) Display a dashed line showing the cannonball's path.

c) Play a sound when the blocker hits the top or bottom of the screen.

d) Play a sound when the target hits the top or bottom of the screen.

e) Enhance the app to have nine levels. In each level, the target should have the same number of target pieces as the level.

f) Keep score. Increase the user's score for each target piece hit by 10 times the current level. Decrease the score by 15 times the current level each time the user hits the blocker. Display the highest score on the screen in the upper-left corner.

g) Save the top five high scores in a `SharedPreferences` file. When the game ends display an `AlertDialog` with the scores shown in descending order. If the user's score is one of the top five, highlight that score by displaying an asterisk (*) next to it.

h) Add an explosion animation each time the cannonball hits one of the target pieces.

i) Add an explosion animation each time the cannonball hits the blocker.

j) When the cannonball hits the blocker, increase the blocker's length by 5%.

k) Make the game more difficult as it progresses by increasing the speed of the target and the blocker.

l) Add multiplayer functionality allowing two users to play on the same device.

m) Increase the number of obstacles between the cannon and the target.

n) Add a bonus round that lasts for four seconds. Change the color of the target and add music to indicate that it is a bonus round. If the user hits a piece of the target during those four seconds, give the user 1000 bonus points.

6.6 *(Brick Game App)* Create a game similar to the cannon game that shoots pellets at a stationary brick wall. The goal is to destroy enough of the wall to shoot the moving target behind it. The faster you break through the wall and get the target, the higher your score. Vary the color of the bricks and the number of shots required to destroy each—for example, red bricks can be destroyed in three shots, yellow bricks can be destroyed in six shots, etc. Include multiple layers to the wall and a small moving target (e.g., an icon, animal, etc.). Keep score. Increase difficulty with each round by adding more layers to the wall and increasing the speed of the moving target.

6.7 *(Tablet App: Multiplayer Horse Race with Cannon Game)* One of the most popular carnival or arcade games is the horse race. Each player is assigned a horse. To move the horse, the players must perform a skill—such as shooting a stream of water at a target. Each time a player hits a target, that player's horse moves forward. The goal is to hit the target as many times as possible and as quickly as possible to move the horse toward the finish line and win the race.

Create a multiplayer tablet app that simulates the **Horse Race** game with two players. Instead of a stream of water, use the **Cannon Game** as the skill that will move each horse. Each time a player hits a target piece with the cannonball, move that player's horse one position to the right.

Set the orientation of the screen to landscape and target API level 11 (Android 3.0) or higher so the game runs on tablets. Split the screen into three sections. The first section should run across the entire width of the top of the screen; this will be the race track. Below the race track, include two sections side-by-side. In each of these sections, include separate **Cannon Game**s. The two players will need to be sitting side-by-side to play this version of the game.

In the race track, include two horses that start on the left and move right toward a finish line at the right-side of the screen. Number the horses "1" and "2."

Include the many sounds of a traditional horse race. You can find free audios online at websites such as `www.audiomicro.com/` or create your own. Before the race, play an audio of the traditional bugle call—the "Call to Post"—that signifies to the horses to take their mark. Include the sound of the shot to start the race, followed by the announcer saying "And they're off!"

6.8 *(Bouncing Ball Game App)* Create a game app in which the user's goal is to prevent a bouncing ball from falling off the bottom of the screen. When the user presses the start button, a ball bounces off the top, left and right sides (the "walls") of the screen. A horizontal bar on the bottom of the screen serves as a paddle to prevent the ball from hitting the bottom of the screen. (The ball can bounce off the paddle, but not the bottom of the screen.) Allow the user to drag the paddle left and right. If the ball hits the paddle, it bounces up, and the game continues. If the ball hits the bottom, the game ends. Decrease the paddle's width every 20 seconds and increase the speed of the ball to make the game more challenging. Consider adding obstacles at random locations.

6.9 *(Stopwatch App)* Create an app that displays a stopwatch on the screen. Include countdown timer functionality.

6.10 *(Lunar Rotation App)* Create an app that displays an earth object at the center of a circle, and a moon object that rotates around the earth object at a constant speed and radius.

6.11 *(Fireworks Designer App)* Create an app that enables the user to create a customized fireworks display. Create a variety of fireworks demonstrations. Then orchestrate the firing of the fireworks for maximum effect. You might synchronize your fireworks with audios or videos. You could overlay the fireworks on a picture.

6.12 *(Animated Towers of Hanoi App)* Every budding computer scientist must grapple with certain classic problems, and the *Towers of Hanoi* (see Fig. 6.21) is one of the most famous. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk.

A third peg is available for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there's little incentive for us to facilitate their efforts.

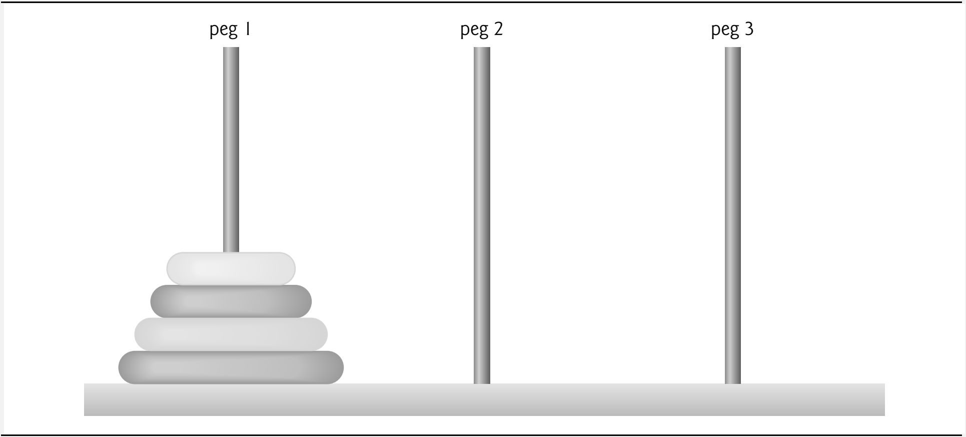


Fig. 6.21 | The Towers of Hanoi for the case with four disks.

Let's assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will display the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving n disks can be viewed in terms of moving only $n - 1$ disks (hence the recursion) as follows:

a) Move $n - 1$ disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
b) Move the last disk (the largest) from peg 1 to peg 3.
c) Move the $n - 1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving $n = 1$ disk (i.e., the base case). This task is accomplished by simply moving the disk, without the need for a temporary holding area.

Write an app to solve the Towers of Hanoi problem. Allow the user to enter the number of disks. Use a recursive `Tower` method with four parameters:

a) the number of disks to be moved,
b) the peg on which these disks are initially threaded,
c) the peg to which this stack of disks is to be moved, and
d) the peg to be used as a temporary holding area.

Your app should display the precise instructions it will take to move the disks from the starting peg to the destination peg and should show animations of the disks moving from peg to peg. For example, to move a stack of three disks from peg 1 to peg 3, your app should display the following series of moves and the corresponding animations:

```
1 --> 3 (This notation means "Move one disk from peg 1 to peg 3.")
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Doodlz App

7

Objectives

In this chapter you'll:

- Detect when the user touches the screen, moves a finger across the screen and removes a finger from the screen.
- Process multiple touches so the user can draw with multiple fingers at once.
- Use a `SensorManager` and the accelerometer to detect motion events.
- Use an `AtomicBoolean` object to allow multiple threads to access a `boolean` value in a thread-safe manner.
- Use a `Paint` object to specify the color and width of a line.
- Use `Path` objects to store each line's data and use a `Canvas` to draw each line into a `BitMap`.
- Create a menu and display menu items on the action bar.
- Use Android 4.4's immersive mode to enable the user to draw on the entire screen.
- Use Android 4.4's printing framework and the Android Support Library class `PrintHelper` to enable the user to print a drawing.

7.1 Introduction

The **Doodlz** app (Fig. 7.1) enables you to paint by dragging one or more fingers across the screen. The app uses Android 4.4's *immersive mode* so that you can draw on the entire screen—the device's *system bars* and *action bar* toggle between displayed and hidden when you tap the screen.

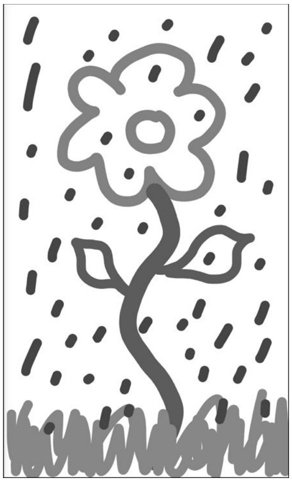


Fig. 7.1 | **Doodlz** app with a finished drawing.

The app's options enable you to set the drawing color and line width. The **Choose Color** dialog (Fig. 7.2(a)) provides *alpha (transparency)*, red, green and blue `SeekBar`s (i.e., sliders) that allow you to select the ARGB color (introduced in Section 1.9). As you move each `SeekBar`'s thumb, the updated color is displayed below the `SeekBar`s. The **Choose Line Width** dialog (Fig. 7.2(b)) provides a single `SeekBar` that controls the thickness of the line that you'll draw. Additional menu items (Fig. 7.3) in the app's *options menu* allow you to turn your finger into an eraser (**Eraser**), to clear the screen (**Clear**), to save the current drawing into your device's **Gallery** (**Save**) and, on Android 4.4 devices, to print the current drawing. Depending on your device's screen size, some or all of the app's menu items are displayed directly on the action bar—any that do not fit are displayed in the options menu. At any point, you can *shake* the device to clear the entire drawing from the screen. You test-drove this app in Section 1.9, so we do not present a test drive in this chapter. Though this app works in AVDs, the capabilities are more fluid on actual devices. [*Note:* Due to a **Gallery** app bug at the time of this writing, on some devices you might need to take a picture with the device's camera app before you'll be able to save properly from the **Doodlz** app.]

a) **Choose Color** dialog

b) **Choose Line Width** dialog

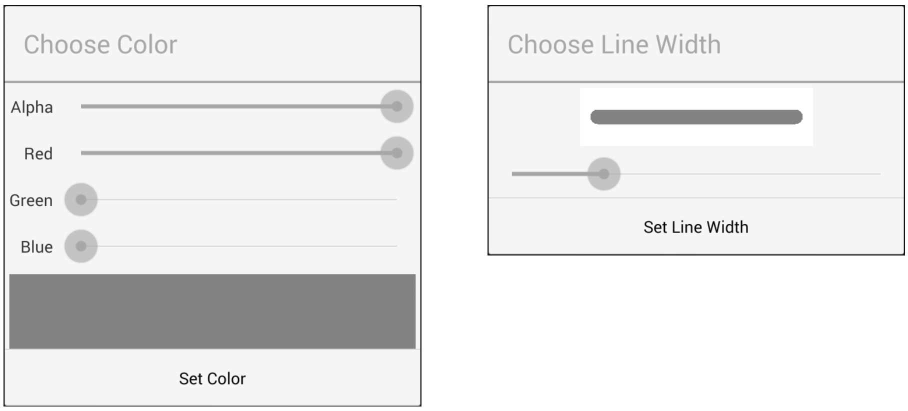


Fig. 7.2 | **Choose Color** and **Choose Line Width** dialogs for the **Doodlz** app.

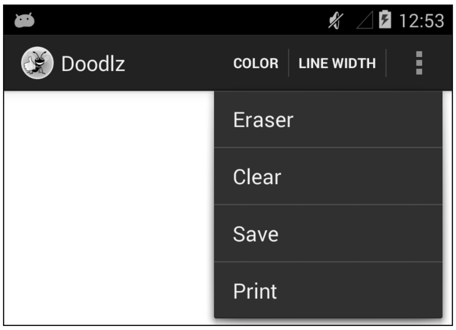


Fig. 7.3 | **Doodlz** app additional menu options as displayed on an Android 4.4 phone.

7.2 Technologies Overview

This section presents the new technologies that we use in the **Doodlz** app.

7.2.1 Using SensorManager to Listen for Accelerometer Events

In this app, you can shake the device to erase the current drawing. Most devices have an **accelerometer** that allows apps to detect movement. Other sensors currently supported by Android include gravity, gyroscope, light, linear acceleration, magnetic field, orientation, pressure, proximity, rotation vector and temperature. The list of **`Sensor`** constants representing these sensor types can be found at:

```
http://developer.android.com/reference/android/hardware/Sensor.html
```

We'll discuss in Section 7.5 the accelerometer and sensor event handling. For a complete discussion of Android's other sensors, see the *Sensors Overview* at

```
http://developer.android.com/guide/topics/sensors/
   sensors_overview.html
```

7.2.2 Custom DialogFragments

Several previous apps have used `AlertDialogs` in `DialogFragments` to display information to the user or to ask questions and receive responses from the user in the form of `Button` clicks. The `AlertDialogs` you've used so far were created using anonymous inner classes that extended `DialogFragment` and displayed only text and buttons. `AlertDialogs` may also contain custom `Views`. In this app, you'll define three subclasses of `DialogFragment`:

- `ColorDialogFragment` (Section 7.7) displays an `AlertDialog` with a custom `View` containing GUI components for previewing and selecting a new ARGB drawing color.
- `LineWidthDialogFragment` (Section 7.8) displays an `AlertDialog` with a custom `View` containing GUI components for previewing and selecting the line thickness.
- `EraseImageDialogFragment` (Section 7.9) displays a standard `AlertDialog` asking the user to confirm whether the entire image should be erased.

For the `ColorDialogFragment` and `EraseImageDialogFragment`, you'll inflate the custom `View` from a layout resource file. In each of the three `DialogFragment` subclasses, you'll also override the following `Fragment` lifecycle methods:

- **`onAttach`**—The first `Fragment` lifecycle method called when a `Fragment` is attached to a parent `Activity`.
- **`onDetach`**—The last `Fragment` lifecycle method called when a `Fragment` is about to be detached from a parent `Activity`.

Preventing Multiple Dialogs from Appearing at the Same Time

It's possible that the event handler for the shake event could try to display the confirmation dialog for erasing an image when another dialog is already on the screen. To prevent this, you'll use `onAttach` and `onDetach` to set the value of a `boolean` that indicates whether a dialog is on the screen. When the `boolean`'s value is `true`, we will not allow the event handler for the shake event to display a dialog.

7.2.3 Drawing with Canvas and `Bitmap`

This app draws lines onto **`Bitmaps`** (package `android.graphics`). You can associate a `Canvas` with a `Bitmap`, then use the `Canvas` to draw on the `Bitmap`, which can then be displayed on the screen (Section 7.6). A `Bitmap` can also be saved into a file—we'll use this capability to store drawings in the device's gallery when you touch the **Save** option.

7.2.4 Processing Multiple Touch Events and Storing Lines in `Paths`

You can drag one or more fingers across the screen to draw. The app stores the information for each *individual* finger as a **`Path`** object (package `android.graphics`) that represents line segments and curves. You process *touch events* by overriding the `View` method **`onTouchEvent`** (Section 7.6). This method receives a **`MotionEvent`** (package `android.view`) that contains the type of touch event that occurred and the ID of the finger (i.e., pointer) that generated the event. We use the IDs to distinguish the different fingers and add information to the corresponding `Path` objects. We use the type of the touch event to determine whether the user has *touched* the screen, *dragged* across the screen or *lifted a finger* from the screen.

7.2.5 Android 4.4 Immersive Mode

Android 4.4 introduces a new full-screen **immersive mode** (Section 7.6) that enables an app to take advantage of the entire screen, but still allows the user to access the system bars when necessary. In this app, you'll use this mode when the app is running on an Android 4.4 or higher device.

7.2.6 `GestureDetector` and `SimpleOnGestureListener`

This app uses a **`GestureDetector`** (package `android.view`) to hide or show the device's system bars and the app's action bar. A `GestureDetector` allows an app to react to user interactions such as *flings*, *single taps*, *double taps*, *long presses* and *scrolls* by implementing the methods of interfaces **`GestureDetector.OnGestureListener`** and **`GestureDetector.OnDoubleTapListener`** interfaces. Class **`GestureDetector.SimpleOnGestureListener`** is an *adapter class* that implements all the methods of these two interfaces, so you can extend this class and override just the method(s) you need from these interfaces. In Section 7.6, you'll initialize a `GestureDetector` with a `SimpleOnGestureListener`, which will handle the *single-tap* event that hides or shows the system bars and action bar.

7.2.7 Saving the Drawing to the Device's Gallery

The app provides a **Save** option that allows the user to save a drawing into the device's gallery—the default location in which photos taken with the device are stored. A **`ContentResolver`** (package `android.content`) enables the app to read data from and store data on a device. You'll use a `ContentResolver` (Section 7.6) and the method **`insertImage`** of class `MediaStore.Images.Media` to save an image into the device's **Gallery**. The **`MediaStore`** manages media files (images, audio and video) stored on a device.

7.2.8 Android 4.4 Printing and the Android Support Library's `PrintHelper` Class

Android 4.4 now includes a printing framework. In this app, we use class `PrintHelper` (Section 7.6) to print the current drawing. Class `PrintHelper` provides a user interface for selecting a printer, has a method for determining whether a given device supports printing and provides a method for printing a `Bitmap`. `PrintHelper` is part of the *Android Support Library*—a set of libraries that are commonly used to provide new Android features for use in older Android versions. The libraries also include additional convenience features, like class `PrintHelper`, that support specific Android versions.

7.3 Building the App's GUI and Resource Files

In this section, you'll create the **Doodlz** app's resource files, GUI layout files and classes.

7.3.1 Creating the Project

Begin by creating a new Android project named `Doodlz`. Specify the following values in the **New Android Project** dialog, then press **Finish**:

- **Application Name:** `Doodlz`
- **Project Name:** `Doodlz`
- **Package Name:** `com.deitel.doodlz`
- **Minimum Required SDK:** `API18: Android 4.3`
- **Target SDK:** `API19: Android 4.4`
- **Compile With:** `API19: Android 4.4`
- **Theme:** `Holo Light with Dark Action Bar`

In the **New Android Project** dialog's second **New Android Application** step, leave the default settings, and press **Next >**. In the **Configure Launcher Icon** step, select an app icon image, then press **Next >**. In the **Create Activity** step, select **Blank Activity**, then press **Next >**. In the **Blank Activity** step, leave the default settings and click **Finish** to create the project. Open `activity_main.xml` in the **Graphical Layout** editor and select **Nexus 4** from the screen-type drop-down list. Once again, we'll use this device as the basis for our design.

The new project will automatically be configured to use the current version of the Android Support Library. If you're updating an existing project, you can add the latest version of the Android Support Library to your project. For details, visit:

```
http://developer.android.com/tools/support-library/index.html
http://developer.android.com/tools/support-library/setup.html
```

7.3.2 `strings.xml`

You created `String` resources in earlier chapters, so we show only a table of the `String` resource names and corresponding values here (Fig. 7.4). Double click `strings.xml` in the `res/values` folder to display the resource editor for creating these `String` resources.

Resource name	Value
`app_name`	`Doodlz`
`button_erase`	`Erase Image`
`button_cancel`	`Cancel`
`button_set_color`	`Set Color`
`button_set_line_width`	`Set Line Width`
`line_imageview_description`	`This displays the line thickness`
`label_alpha`	`Alpha`
`label_red`	`Red`
`label_green`	`Green`
`label_blue`	`Blue`
`menuitem_clear`	`Clear`
`menuitem_color`	`Color`
`menuitem_eraser`	`Eraser`
`menuitem_line_width`	`Line Width`
`menuitem_save`	`Save`
`menuitem_print`	`Print`
`message_erase`	`Erase the drawing?`
`message_error_saving`	`There was an error saving the image`
`message_saved`	`Your painting has been saved to the Gallery`
`message_error_printing`	`Your device does not support printing`
`title_color_dialog`	`Choose Color`
`title_line_width_dialog`	`Choose Line Width`

Fig. 7.4 | `String` resources used in the **Doodlz** app.

7.3.3 `dimens.xml`

Figure 7.5 shows a table of the dimension resource names and values that we added to `dimens.xml`. Open `dimens.xml` in the `res/values` folder to display the resource editor for creating these resources. The `line_imageview_height` resource specifies the height of the `ImageView` that previews the line width in the `LineWidthDialogFragment`, and the `color_view_height` resource specifies height of the `View` that previews the drawing color in the `ColorDialogFragment`.

Resource name	Value
`line_imageview_height`	`50dp`
`color_view_height`	`80dp`

Fig. 7.5 | Dimension resources used in the **Doodlz** app.

7.3.4 Menu for the `DoodleFragment`

In Chapter 5, you used the default menu provided by the IDE to display the **Flag Quiz** app's **Settings** menu item. You will not use the default menu in this app, so you can delete the `main.xml` file in your project's `res/menu` folder. In this app, you'll define your own menu for the `DoodleFragment`.

Menus for Different Android Versions

You'll provide two versions of the `DoodleFragment`'s menu—one for Android 4.3 and earlier devices and one for Android 4.4 and higher devices. Printing is available only in Android 4.4 and higher, so only the menu for such devices will include a **Print** option. To support separate menus, you'll define one menu resource in the `res/menu` folder and a separate menu resource in the `res/menu-v19` folder—19 is the Android API version that corresponds to Android 4.4. Android will choose the menu resource in the `res/menu-v19` folder when the app is running on Android 4.4 and higher devices. To create the `res/menu-v19` folder, right click the `res` folder, select **New > Folder**, specify the **Folder name** `menu-v19` and click **Finish**.

Menu for Android 4.3 and Earlier Versions

To create the menu resource for Android 4.3 and earlier versions:

1. Right click the `res/menu` folder and select **New > Android XML File**.
2. In the dialog that appears, name the file `doodle_fragment_menu.xml` and click **Finish**. The IDE opens the file in the editor for menu resources.
3. Click **Add...**, click the editor's **Layout** tab in the dialog that appears, select **Item** and click **OK**. The IDE highlights the new item and displays its attributes to the right.
4. Change its **Id** to `@+id/color`, its **Title** to `@string/menuitem_color` and its **Show as action** to `ifRoom`. The value `ifRoom` indicates that Android should display the menu item on the action bar if there's room available; otherwise, the menu item will appear in the options menu at the right side of the action bar. Other **Show as action** values can be found at `http://developer.android.com/guide/topics/resources/menu-resource.html`.
5. Repeat Steps 3 and 4 for the `lineWidth`, `eraser`, `clear` and `save` items in Fig. 7.6. Note that when you click **Add...** for each additional menu item, you'll need to select **Create a new element at the top level in Menu** in the dialog that appears.
6. Save and close `doodle_fragment_menu.xml`.

Id	Title
`@+id/lineWidth`	`@string/menuitem_line_width`
`@+id/eraser`	`@string/menuitem_eraser`
`@+id/clear`	`@string/menuitem_clear`
`@+id/save`	`@string/menuitem_save`

Fig. 7.6 | Additional menu items for the `DoodleFragment`.

Menu for Android 4.4 and Higher Versions
To create the menu resource for Android 4.4 and higher devices:

1. Copy `doodle_fragment_menu.xml` from `res/menu`, paste it into `res/menu-v19` and open the file.
2. Click **Add...**, select **Create a new element at the top level in Menu** in the dialog that appears, then select **Item** and click **OK**.
3. Change the new item's **Id** to `@+id/print`, its **Title** to `@string/menuitem_print` and its **Show as action** to `ifRoom`.

7.3.5 `activity_main.xml` Layout for `MainActivity`

The `activity_main.xml` layout for this app's `MainActivity` contains only the `DoodleFragment`. To add this `Fragment` to the layout:

1. Open `activity_main.xml` in the **Graphical Layout** editor, then follow the steps in Section 2.5.2 to switch from a `FrameLayout` to a `RelativeLayout`.
2. From the **Palette**'s **Layouts** section, drag a **Fragment** onto the design area or onto the `RelativeLayout` node in the **Outline** window.
3. The preceding step displays the **Choose Fragment Class** dialog. Click **Create New...** to display the **New Java Class** dialog.
4. Enter `DoodleFragment` in the dialog's **Name** field, change the **Superclass** field's value to `android.app.Fragment` and click **Finish** to create the class. The IDE opens the Java file for the class, which you can close for now.
5. Change the new `Fragment`'s **Id** to `@+id/doodleFragment`, then save the layout.

7.3.6 `fragment_doodle.xml` Layout for `DoodleFragment`

The `fragment_doodle.xml` layout for the `DoodleFragment` contains a `FrameLayout` that displays the `DoodleView`. In this section, you'll create `DoodleFragment`'s layout and the `DoodleView` class. To add the `fragment_doodle.xml` layout:

1. Expand the project's `res/layout` node in the **Package Explorer**.
2. Right click the `layout` folder and select **New > Android XML File** to display the **New Android XML File** dialog.
3. In the dialog's **File** field, enter `fragment_doodle.xml`
4. In the **Root Element** section, select **FrameLayout**, then click **Finish**.
5. From the **Palette**'s **Advanced** section, drag a **view** (with a lowercase **v**) onto the GUI.
6. The previous step displays the **Choose Custom View Class** dialog. In that dialog, click **Create New...** to display the **New Java Class** dialog.
7. In the **Name** field, enter `DoodleView`. Ensure that **Constructors from superclass** is checked, then click **Finish**. This creates and opens `DoodleView.java`. We'll be using only the two-argument constructor, so delete the other two. Save and close `DoodleView.java`.
8. In `fragment_doodle.xml`, select **view1** in the **Outline** window. In the **Properties** window's **Layout Parameters** section, set **Width** and **Height** to `match_parent`.

9. In the **Outline** window, right click **view1**, select **Edit ID...**, rename `view1` as `doodleView` and click **OK**.
10. Save and close `fragment_doodle.xml`.

7.3.7 `fragment_color.xml` Layout for `ColorDialogFragment`

The `fragment_color.xml` layout for the `ColorDialogFragment` contains a `GridLayout` that displays a GUI for selecting and previewing a new drawing color. In this section, you'll create `ColorDialogFragment`'s layout and the `ColorDialogFragment` class. To add the `fragment_color.xml` layout:

1. Expand the project's `res/layout` node in the **Package Explorer**.
2. Right click the `layout` folder and select **New > Android XML File** to display the **New Android XML File** dialog.
3. In the dialog's **File** field, enter `fragment_color.xml`
4. In the **Root Element** section, select **GridLayout**, then click **Finish**.
5. In the **Outline** window, select the **GridLayout** and change its **Id** value to `@+id/colorDialogGridLayout`.
6. Using the **Graphical Layout** editor's **Palette**, drag **TextView**s, **SeekBar**s and a **View** onto the `colorDialogGridLayout` node in the **Outline** window. Drag the items in the order they're listed in Fig. 7.7 and set each item's **Id** as shown in the figure.

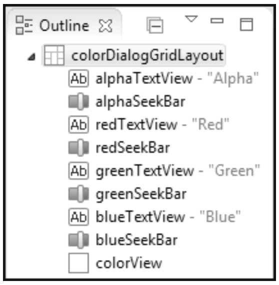

Outline
colorDialogGridLayout
alphaTextView - "Alpha"
alphaSeekBar
redTextView - "Red"
redSeekBar
greenTextView - "Green"
greenSeekBar
blueTextView - "Blue"
blueSeekBar
colorView

Fig. 7.7 | **Outline** view for `fragment_color.xml`.

7. After completing Step 6, configure the GUI component properties with the values shown in Fig. 7.8, then save and close `fragment_color.xml`.

GUI component	Property	Value

`colorDialogGridLayout`	**Column Count**
	Orientation
	Use Default Margins
		`2`
		`vertical`
		`true`

Fig. 7.8 | Property values for the GUI components in `fragment_color.xml`. (Part 1 of 3.)

GUI component	Property	Value
`alphaTextView`	*Layout Parameters*	
	Column	`0`
	Gravity	`right|center_vertical`
	Row	`0`
	Other Properties	
	Text	`@string/label_alpha`
`alphaSeekBar`	*Layout Parameters*	
	Column	`1`
	Gravity	`fill_horizontal`
	Row	`0`
	Other Properties	
	Max	`255`
`redTextView`	*Layout Parameters*	
	Column	`0`
	Gravity	`right|center_vertical`
	Row	`1`
	Other Properties	
	Text	`@string/label_red`
`redSeekBar`	*Layout Parameters*	
	Column	`1`
	Gravity	`fill_horizontal`
	Row	`1`
	Other Properties	
	Max	`255`
`greenTextView`	*Layout Parameters*	
	Column	`0`
	Gravity	`right|center_vertical`
	Row	`2`
	Other Properties	
	Text	`@string/label_green`
`greenSeekBar`	*Layout Parameters*	
	Column	`1`
	Gravity	`fill_horizontal`
	Row	`2`
	Other Properties	
	Max	`255`
`blueTextView`	*Layout Parameters*	
	Column	`0`
	Gravity	`right|center_vertical`
	Row	`3`
	Other Properties	
	Text	`@string/label_blue`

Fig. 7.8 | Property values for the GUI components in `fragment_color.xml`. (Part 2 of 3.)

GUI component	Property	Value
`blueSeekBar`	*Layout Parameters*	
	Column	`1`
	Gravity	`fill_horizontal`
	Row	`3`
	Other Properties	
	Max	`255`
`colorView`	*Layout Parameters*	
	Height	`@dimen/color_view_height`
	Column	`0`
	Column Span	`2`
	Gravity	`fill_horizontal`

Fig. 7.8 | Property values for the GUI components in `fragment_color.xml`. (Part 3 of 3.)

Adding Class **`ColorDialogFragment`** *to the Project*
To add class `ColorDialogFragment` to the project:

1. Right click the package `com.deitel.doodlz` in the project's `src` folder and select **New > Class** to display the **New Java Class** dialog.
2. In the **Name** field, enter `ColorDialogFragment`.
3. In the **Superclass** field, change the superclass to `android.app.DialogFragment`.
4. Click **Finish** to create the class.

7.3.8 `fragment_line_width.xml` Layout for `LineWidthDialogFragment`

The `fragment_line_width.xml` layout for the `LineWidthDialogFragment` contains a `GridLayout` that displays a GUI for selecting and previewing a new line thickness. In this section, you'll create `LineWidthDialogFragment`'s layout and the `LineWidthDialogFragment` class. To add the `fragment_line_width.xml` layout:

1. Expand the project's `res/layout` node in the **Package Explorer**.
2. Right click the `layout` folder and select **New > Android XML File** to display the **New Android XML File** dialog.
3. In the dialog's **File** field, enter `fragment_line_width.xml`
4. In the **Root Element** section, select **GridLayout**, then click **Finish**.
5. In the **Outline** window, select the **GridLayout** and change its **Id** value to `@+id/lineWidthDialogGridLayout`.
6. Using the **Graphical Layout** editor's **Palette**, drag an **ImageView** and a **SeekBar** onto the `lineWidthDialogGridLayout` node in the **Outline** window so that the window appears as shown in Fig. 7.9. Set each item's **Id** as shown in the figure.
7. After completing Step 6, configure the GUI component properties with the values shown in Fig. 7.10, then save and close `fragment_line_width.xml`.

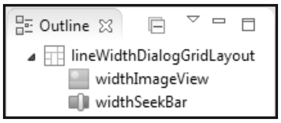


Fig. 7.9 | **Outline** view for `fragment_line_width.xml`.

GUI component	Property	Value
`lineWidthDialog-GridLayout`	**Column Count**	`1`
	Orientation	`vertical`
	Use Default Margins	`true`
`widthImageView`	*Layout Parameters*	
	Height	`@dimen/line_imageview_height`
	Gravity	`fill_horizontal`
	Other Properties	
	Content Description	`@string/line_imageview_description`
`widthSeekBar`	*Layout Parameters*	
	Gravity	`fill_horizontal`
	Other Properties	
	Max	`50`

Fig. 7.10 | Property values for the GUI components in `fragment_line_width.xml`.

Adding Class `LineWidthDialogFragment` to the Project

To add class `LineWidthDialogFragment` to the project:

1. Right click the package `com.deitel.doodlz` in the project's `src` folder and select **New > Class** to display the **New Java Class** dialog.
2. In the **Name** field, enter `LineWidthDialogFragment`.
3. In the **Superclass** field, change the superclass to `android.app.DialogFragment`.
4. Click **Finish** to create the class.

7.3.9 Adding Class `EraseImageDialogFragment`

The `EraseImageDialogFragment` does not require a layout resource as it will display a simple `AlertDialog` containing text. To add class `EraseImageDialogFragment` to the project:

1. Right click the package `com.deitel.doodlz` in the project's `src` folder and select **New > Class** to display the **New Java Class** dialog.
2. In the **Name** field, enter `EraseImageDialogFragment`.
3. In the **Superclass** field, change the superclass to `android.app.DialogFragment`.
4. Click **Finish** to create the class.

7.4 MainActivity Class

This app consists of six classes:

- MainActivity (Fig. 7.11)—Serves as the parent Activity for this app's Fragments.
- DoodleFragment (Section 7.5)—Manages the DoodleView and accelerometer event handling.
- DoodleView (Section 7.6)—Provides the drawing, saving and printing capabilities.
- ColorDialogFragment (Section 7.7)—A DialogFragment that's displayed when the user taps **COLOR** to set the drawing color.
- LineWidthDialogFragment (Section 7.8)—A DialogFragment that's displayed when the user taps **LINE WIDTH** to set the line width.
- EraseImageDialogFragment (Section 7.9)—A DialogFragment that's displayed when the user taps **CLEAR** or shakes the device to erase the current drawing.

Class MainActivity's onCreate method (Fig. 7.11) inflates the GUI (line 16), then uses the techniques you learned in Section 5.2.2 to determine the device's size and set MainActivity's orientation. If this app is running on an extra large device (line 24), we set the orientation to landscape (lines 25–26); otherwise, we set it to portrait (lines 28–29).

```
 1  // MainActivity.java
 2  // Sets MainActivity's layout
 3  package com.deitel.doodlz;
 4
 5  import android.app.Activity;
 6  import android.content.pm.ActivityInfo;
 7  import android.content.res.Configuration;
 8  import android.os.Bundle;
 9
10  public class MainActivity extends Activity
11  {
12     @Override
13     protected void onCreate(Bundle savedInstanceState)
14     {
15        super.onCreate(savedInstanceState);
16        setContentView(R.layout.activity_main);
17
18        // determine screen size
19        int screenSize =
20           getResources().getConfiguration().screenLayout &
21           Configuration.SCREENLAYOUT_SIZE_MASK;
22
23        // use landscape for extra large tablets; otherwise, use portrait
24        if (screenSize == Configuration.SCREENLAYOUT_SIZE_XLARGE)
25           setRequestedOrientation(
26              ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
```

Fig. 7.11 | MainActivity class. (Part 1 of 2.)

```
      else
         setRequestedOrientation(
            ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);
   }
} // end class MainActivity
```

Fig. 7.11 | `MainActivity` class. (Part 2 of 2.)

7.5 `DoodleFragment` Class

Class `DoodleFragment` (Figs. 7.12–7.19) displays the `DoodleView` (Section 7.6), manages the menu options displayed on the action bar and in the options menu and manages the sensor event handling for the app's *shake-to-erase* feature.

`package` *Statement,* `import` *Statements and Fields*

Section 7.2 discussed the key new classes and interfaces that class `DoodleFragment` uses. We've highlighted these classes and interfaces in Fig. 7.12. `DoodleView` variable `doodleView` (line 22) represents the drawing area. The `float` variables declared in lines 23–25 are used to calculate changes in the device's acceleration to determine when a *shake event* occurs (so we can ask whether the user would like to erase the drawing), and the constant in line 29 is used to ensure that small movements are *not* interpreted as shakes—we picked this constant via trial and error by shaking the app on several devices. Line 26 defines a `boolean` variable with the default value `false` that will be used throughout this class to specify when there's a dialog displayed on the screen. We use this to prevent multiple dialogs from being displayed at the same time—for example, if the **Choose Color** dialog is displayed and the user accidentally shakes the device, the dialog for erasing the image should *not* be displayed.

```
 1  // DoodleFragment.java
 2  // Fragment in which the DoodleView is displayed
 3  package com.deitel.doodlz;
 4
 5  import android.app.Fragment;
 6  import android.content.Context;
 7  import android.graphics.Color;
 8  import android.hardware.Sensor;
 9  import android.hardware.SensorEvent;
10  import android.hardware.SensorEventListener;
11  import android.hardware.SensorManager;
12  import android.os.Bundle;
13  import android.view.LayoutInflater;
14  import android.view.Menu;
15  import android.view.MenuInflater;
16  import android.view.MenuItem;
17  import android.view.View;
```

Fig. 7.12 | `DoodleFragment` class `package` statement, `import` statements and fields. (Part 1 of 2.)

```
import android.view.ViewGroup;

public class DoodleFragment extends Fragment
{
   private DoodleView doodleView; // handles touch events and draws
   private float acceleration;
   private float currentAcceleration;
   private float lastAcceleration;
   private boolean dialogOnScreen = false;

   // value used to determine whether user shook the device to erase
   private static final int ACCELERATION_THRESHOLD = 100000;

```

Fig. 7.12 | `DoodleFragment` class `package` statement, `import` statements and fields. (Part 2 of 2.)

Overriding `Fragment` *Method* `onCreateView`

Method `onCreateView` (Fig. 7.13) inflates the `DoodleFragment`'s GUI and initializes the instance variables. Like an `Activity`, a `Fragment` can place items in the app's action bar and options menu. To do so, the `Fragment` must call its `setHasOptionsMenu` method with the argument `true`. If the parent `Activity` also has options menu items, then both the `Activity`'s and the `Fragment`'s items will be placed on the action bar and in the options menu (based on their settings).

```
31    // called when Fragment's view needs to be created
32    @Override
33    public View onCreateView(LayoutInflater inflater, ViewGroup container,
34       Bundle savedInstanceState)
35    {
36       super.onCreateView(inflater, container, savedInstanceState);
37       View view =
38          inflater.inflate(R.layout.fragment_doodle, container, false);
39
40       setHasOptionsMenu(true); // this fragment has menu items to display
41
42       // get reference to the DoodleView
43       doodleView = (DoodleView) view.findViewById(R.id.doodleView);
44
45       // initialize acceleration values
46       acceleration = 0.00f;
47       currentAcceleration = SensorManager.GRAVITY_EARTH;
48       lastAcceleration = SensorManager.GRAVITY_EARTH;
49       return view;
50    }
51
```

Fig. 7.13 | Overriding `Fragment` method `onCreateView`.

Line 43 gets a reference to the `DoodleView`, then lines 46–48 initialize the instance variables that help calculate acceleration changes to determine whether the user shook the device. We initially set variables currentAcceleration and lastAcceleration to SensorManager's GRAVITY_EARTH constant, which represents the acceleration due to gravity on earth. SensorManager also provides constants for other planets in the solar system, for the moon and for several other entertaining values, which you can see at:

```
http://developer.android.com/reference/android/hardware/
   SensorManager.html
```

Methods **onStart** *and* **enableAccelerometerListening**

Accelerometer listening should be enabled only when the DoodleFragment is on the screen. For this reason, we override Fragment lifecycle method **onStart** (Fig. 7.14, lines 53–58), which calls method enableAccelerometerListening (lines 61–72) to begin listening for accelerometer events. A SensorManager is used to register listeners for accelerometer events.

```
52      // start listening for sensor events
53      @Override
54      public void onStart()
55      {
56         super.onStart();
57         enableAccelerometerListening(); // listen for shake
58      }
59
60      // enable listening for accelerometer events
61      public void enableAccelerometerListening()
62      {
63         // get the SensorManager
64         SensorManager sensorManager =
65            (SensorManager) getActivity().getSystemService(
66               Context.SENSOR_SERVICE);
67
68         // register to listen for accelerometer events
69         sensorManager.registerListener(sensorEventListener,
70            sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
71            SensorManager.SENSOR_DELAY_NORMAL);
72      }
73
```

Fig. 7.14 | Methods onStart and enableAccelerometerListening.

Method enableAccelerometerListening first uses Activity's getSystemService method to retrieve the system's SensorManager service, which enables the app to interact with the device's sensors. Lines 69–71 then register to receive accelerometer events using SensorManager's **registerListener** method, which receives three arguments:

- The SensorEventListener that responds to the events (defined in Fig. 7.16)
- A Sensor object representing the type of sensor data the app wishes to receive—this is retrieved by calling SensorManager's **getDefaultSensor** method and passing a Sensor-type constant (Sensor.TYPE_ACCELEROMETER in this app).
- A rate at which sensor events should be delivered to the app. We chose SENSOR_DELAY_NORMAL to receive sensor events at the default rate—a faster rate can be used to get more accurate data, but this is also more CPU and battery intensive.

Methods `onPause` *and* `disableAccelerometerListening`

To ensure that accelerometer listening is disabled when the `DoodleFragment` is not on the screen, we override `Fragment` lifecycle method `onPause` (Fig. 7.15, lines 75–80), which calls method `disableAccelerometerListening` (lines 83–93). Method `disableAccelerometerListening` uses class `SensorManager`'s **`unregisterListener`** method to stop listening for accelerometer events.

```
74      // stop listening for sensor events
75      @Override
76      public void onPause()
77      {
78         super.onPause();
79         disableAccelerometerListening(); // stop listening for shake
80      }
81
82      // disable listening for accelerometer events
83      public void disableAccelerometerListening()
84      {
85         // get the SensorManager
86         SensorManager sensorManager =
87            (SensorManager) getActivity().getSystemService(
88               Context.SENSOR_SERVICE);
89
90         // stop listening for accelerometer events
91         sensorManager.unregisterListener(sensorEventListener,
92            sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER));
93      }
94
```

Fig. 7.15 | Methods `onPause` and `disableAccelerometerListening`.

Anonymous Inner Class That Implements Interface `SensorEventListener` *to Process Accelerometer Events*

Figure 7.16 overrides `SensorEventListener` method **`onSensorChanged`** (lines 100–125) to process accelerometer events. If the user moves the device, this method determines whether the movement was enough to be considered a shake. If so, line 123 calls method `confirmErase` (Fig. 7.17) to display an `EraseImageDialogFragment` (Section 7.9) and confirm whether the user really wants to erase the image. Interface `SensorEventListener` also contains method `onAccuracyChanged` (lines 128–131)—we don't use this method in this app, so we provide an empty body because the method is required by the interface.

```
95      // event handler for accelerometer events
96      private SensorEventListener sensorEventListener =
97         new SensorEventListener()
98         {
```

Fig. 7.16 | Anonymous inner class that implements interface `SensorEventListener` to process accelerometer events. (Part 1 of 2.)

```
 99          // use accelerometer to determine whether user shook device
100          @Override
101          public void onSensorChanged(SensorEvent event)
102          {
103             // ensure that other dialogs are not displayed
104             if (!dialogOnScreen)
105             {
106                // get x, y, and z values for the SensorEvent
107                float x = event.values[0];
108                float y = event.values[1];
109                float z = event.values[2];
110
111                // save previous acceleration value
112                lastAcceleration = currentAcceleration;
113
114                // calculate the current acceleration
115                currentAcceleration = x * x + y * y + z * z;
116
117                // calculate the change in acceleration
118                acceleration = currentAcceleration *
119                   (currentAcceleration - lastAcceleration);
120
121                // if the acceleration is above a certain threshold
122                if (acceleration > ACCELERATION_THRESHOLD)
123                   confirmErase();
124             }
125          } // end method onSensorChanged
126
127          // required method of interface SensorEventListener
128          @Override
129          public void onAccuracyChanged(Sensor sensor, int accuracy)
130          {
131          }
132       }; // end anonymous inner class
133
```

Fig. 7.16 | Anonymous inner class that implements interface `SensorEventListener` to process accelerometer events. (Part 2 of 2.)

The user can shake the device even when dialogs are already displayed on the screen. For this reason, `onSensorChanged` first checks whether a dialog is displayed (line 104). This test ensures that no other dialogs are displayed; otherwise, `onSensorChanged` simply returns. This is important because the sensor events occur in a different thread of execution. Without this test, we'd be able to display the confirmation dialog for erasing the image when another dialog is on the screen.

The **`SensorEvent`** parameter contains information about the sensor change that occurred. For accelerometer events, this parameter's `values` array contains three elements representing the acceleration (in *meter/second*^{2}) in the *x* (left/right), *y* (up/down) and *z* (forward/backward) directions. A description and diagram of the coordinate system used by the `SensorEvent` API is available at:

```
developer.android.com/reference/android/hardware/SensorEvent.html
```

This link also describes the real-world meanings for a `SensorEvent`'s *x*, *y* and *z* values for each different `Sensor`.

Lines 107–109 store the acceleration values. It's important to handle sensor events quickly or to copy the event data (as we did here) because the array of sensor values is *reused* for each sensor event. Line 112 stores the last value of `currentAcceleration`. Line 115 sums the squares of the `x`, `y` and `z` acceleration values and stores them in `currentAcceleration`. Then, using the `currentAcceleration` and `lastAcceleration` values, we calculate a value (`acceleration`) that can be compared to our `ACCELERATION_THRESHOLD` constant. If the value is greater than the constant, the user moved the device enough for this app to consider the movement a shake. In this case, we call method `confirmErase`.

Method **`confirmErase`**

Method `confirmErase` (Fig. 7.17) simply creates an `EraseImageDialogFragment` (Section 7.9) and uses the `DialogFragment` method `show` to display it.

```
134     // confirm whether image should be erased
135     private void confirmErase()
136     {
137        EraseImageDialogFragment fragment = new EraseImageDialogFragment();
138        fragment.show(getFragmentManager(), "erase dialog");
139     }
140
```

Fig. 7.17 | Method `confirmErase` displays an `EraseImageDialogFragment`.

Overridden **`Fragment`** *Methods* **`onCreateOptionsMenu`** *and* **`onOptionsItemSelected`**

Figure 7.18 overrides `Fragment`'s **`onCreateOptionsMenu`** method (lines 142–147) to add the options to the method's `Menu` argument using the method's `MenuInflater` argument. When the user selects a menu item, `Fragment` method **`onOptionsItemSelected`** (lines 150–180) responds to the selection.

```
141     // display this fragment's menu items
142     @Override
143     public void onCreateOptionsMenu(Menu menu, MenuInflater inflater)
144     {
145        super.onCreateOptionsMenu(menu, inflater);
146        inflater.inflate(R.menu.doodle_fragment_menu, menu);
147     }
148
149     // handle choice from options menu
150     @Override
151     public boolean onOptionsItemSelected(MenuItem item)
152     {
```

Fig. 7.18 | Overridden `Fragment` methods `onCreateOptionsMenu` and `onOptionsItemSelected`. (Part 1 of 2.)

```
153       // switch based on the MenuItem id
154       switch (item.getItemId())
155       {
156          case R.id.color:
157             ColorDialogFragment colorDialog = new ColorDialogFragment();
158             colorDialog.show(getFragmentManager(), "color dialog");
159             return true; // consume the menu event
160          case R.id.lineWidth:
161             LineWidthDialogFragment widthdialog =
162                new LineWidthDialogFragment();
163             widthdialog.show(getFragmentManager(), "line width dialog");
164             return true; // consume the menu event
165          case R.id.eraser:
166             doodleView.setDrawingColor(Color.WHITE); // line color white
167             return true; // consume the menu event
168          case R.id.clear:
169             confirmErase(); // confirm before erasing image
170             return true; // consume the menu event
171          case R.id.save:
172             doodleView.saveImage(); // save the current image
173             return true; // consume the menu event
174          case R.id.print:
175             doodleView.printImage(); // print the current images
176             return true; // consume the menu event
177       } // end switch
178
179       return super.onOptionsItemSelected(item); // call super's method
180    } // end method onOptionsItemSelected
181
```

Fig. 7.18 | Overridden `Fragment` methods `onCreateOptionsMenu` and `onOptionsItemSelected`. (Part 2 of 2.)

We use the `MenuItem` argument's **`getItemID`** method (line 154) to get the resource ID of the selected menu item, then take different actions based on the selection. The actions are as follows:

- For `R.id.color`, lines 157–158 create and show a `ColorDialogFragment` (Section 7.7) to allow the user to select a new drawing color.
- For `R.id.lineWidth`, lines 161–163 create and show a `LineWidthDialogFragment` (Section 7.8) to allow the user to select a new drawing color.
- For `R.id.eraser`, line 166 sets the `doodleView`'s drawing color to white, which effectively turns the user's fingers into *erasers*.
- For `R.id.clear`, line 169 calls method `confirmErase` (Fig. 7.17) to display an `EraseImageDialogFragment` (Section 7.9) and confirm whether the user really wants to erase the image.
- For `R.id.save`, line 172 calls `doodleView`'s `saveImage` method to save the painting as an image stored in the device's **Gallery**.
- For `R.id.print`, line 175 calls `doodleView`'s `printImage` method to allow the user to save the image as a PDF or to print the image.

Methods **`getDoodleView`** *and* **`setDialogOnScreen`**

Methods `getDoodleView` and `setDialogOnScreen` (Fig. 7.19) are called by methods of the app's `DialogFragment` subclasses. Method `getDoodleView` returns a reference to this `Fragment`'s `DoodleView` so that a `DialogFragment` can set the drawing color, set the line width or clear the image. Method `setDialogOnScreen` is called by `Fragment` lifecycle methods of the app's `DialogFragment` subclasses to indicate when a dialog is on the screen.

```
    // returns the DoodleView
    public DoodleView getDoodleView()
    {
       return doodleView;
    }

    // indicates whether a dialog is displayed
    public void setDialogOnScreen(boolean visible)
    {
       dialogOnScreen = visible;
    }
}
```

Fig. 7.19 | Methods `getDoodleView` and `setDialogOnScreen`.

7.6 `DoodleView` Class

The `DoodleView` class (Figs. 7.20–7.33) processes the user's touches and draws the corresponding lines.

***`DooldleView`* package** *Statement and* **`import`** *Statements*

Figure 7.20 lists class `DoodleView`'s `package` statement, `import` statements and fields. The new classes and interfaces are highlighted here. Many of these were discussed in Section 7.2 and the rest are discussed as we use them throughout class `DoodleView`.

```
// DoodleView.java
// Main View for the Doodlz app.
package com.deitel.doodlz;

import java.util.HashMap;
import java.util.Map;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Path;
import android.graphics.Point;
import android.os.Build;
import android.provider.MediaStore;
import android.support.v4.print.PrintHelper;
```

Fig. 7.20 | `DooldleView` `package` statement and `import` statements. (Part 1 of 2.)

```
18 import android.util.AttributeSet;
19 import android.view.GestureDetector;
20 import android.view.GestureDetector.SimpleOnGestureListener;
21 import android.view.Gravity;
22 import android.view.MotionEvent;
23 import android.view.View;
24 import android.widget.Toast;
25
```

Fig. 7.20 | `DooldleView` `package` statement and `import` statements. (Part 2 of 2.)

`DoodleView` `static` and Instance Variables

Class `DoodleView`'s `static` and instance variables (Fig. 7.21, lines 30–43) are used to manage the data for the set of lines that the user is currently drawing and to draw those lines. Line 38 creates the `pathMap`, which maps each finger ID (known as a pointer) to a corresponding `Path` object for the lines currently being drawn. Lines 39–40 create the `previousPointMap`, which maintains the last point for each finger—as each finger moves, we draw a line from its current point to its previous point. We discuss the other fields as we use them in class `DoodleView`.

```
26 // the main screen that is painted
27 public class DoodleView extends View
28 {
29    // used to determine whether user moved a finger enough to draw again
30    private static final float TOUCH_TOLERANCE = 10;
31
32    private Bitmap bitmap; // drawing area for display or saving
33    private Canvas bitmapCanvas; // used to draw on bitmap
34    private final Paint paintScreen; // used to draw bitmap onto screen
35    private final Paint paintLine; // used to draw lines onto bitmap
36
37    // Maps of current Paths being drawn and Points in those Paths
38    private final Map<Integer, Path> pathMap = new HashMap<Integer, Path>();
39    private final Map<Integer, Point> previousPointMap =
40       new HashMap<Integer, Point>();
41
42    // used to hide/show system bars
43    private GestureDetector singleTapDetector;
44
```

Fig. 7.21 | `DoodleView` `static` and instance variables.

`DoodleView` Constructor

The constructor (Fig. 7.22) initializes several of the class's instance variables—the two `Maps` are initialized in their declarations in Fig. 7.21. Line 49 creates the `Paint` object `paintScreen` that will be used to display the user's drawing on the screen and line 52 creates the `Paint` object `paintLine` that specifies the settings for the line(s) the user is currently drawing. Lines 53–57 specify the settings for the `paintLine` object. We pass `true` to `Paint`'s **`setAntiAlias`** method to enable *anti-aliasing* which smooths the edges of the lines. Next, we set the `Paint`'s style to `Paint.Style.STROKE` with `Paint`'s **`setStyle`** method. The style can be `STROKE`, `FILL` or `FILL_AND_STROKE` for a line, a filled shape without a border and a filled shape with a border, respectively. The default option is `Paint.Style.FILL`. We set the line's width using `Paint`'s `setStrokeWidth` method. This sets the app's *default line width* to five pixels. We also use `Paint`'s `setStrokeCap` method to round the ends of the lines with `Paint.Cap.ROUND`. Lines 60–61 create a `GestureDetector` that uses the `singleTapListener` to check for single-tap events.

```
45      // DoodleView constructor initializes the DoodleView
46      public DoodleView(Context context, AttributeSet attrs)
47      {
48         super(context, attrs); // pass context to View's constructor
49         paintScreen = new Paint(); // used to display bitmap onto screen
50
51         // set the initial display settings for the painted line
52         paintLine = new Paint();
53         paintLine.setAntiAlias(true); // smooth edges of drawn line
54         paintLine.setColor(Color.BLACK); // default color is black
55         paintLine.setStyle(Paint.Style.STROKE); // solid line
56         paintLine.setStrokeWidth(5); // set the default line width
57         paintLine.setStrokeCap(Paint.Cap.ROUND); // rounded line ends
58
59         // GestureDetector for single taps
60         singleTapDetector =
61            new GestureDetector(getContext(), singleTapListener);
62      }
63
```

Fig. 7.22 | `DoodleView` constructor.

Overridden `View` Method `onSizeChanged`

The `DoodleView`'s size is not determined until it's inflated and added to the `MainActivity`'s `View` hierarchy; therefore, we can't determine the size of the drawing `Bitmap` in `onCreate`. So, we override `View` method `onSizeChanged` (Fig. 7.23), which is called when the `DoodleView`'s size changes—e.g., when it's added to an `Activity`'s `View` hierarchy or when the user rotates the device. In this app, `onSizeChanged` is called only when the `DoodleView` is added to the `Doodlz` `Activity`'s `View` hierarchy, because the app always displays in *portrait* on phones and small tablets, and in *landscape* on large tablets.

```
   // Method onSizeChanged creates Bitmap and Canvas after app displays
   @Override
   public void onSizeChanged(int w, int h, int oldW, int oldH)
   {
      bitmap = Bitmap.createBitmap(getWidth(), getHeight(),
         Bitmap.Config.ARGB_8888);
      bitmapCanvas = new Canvas(bitmap);
      bitmap.eraseColor(Color.WHITE); // erase the Bitmap with white
   }

```

Fig. 7.23 | Overridden `View` method `onSizeChanged`.

Bitmap's static **createBitmap** method creates a Bitmap of the specified width and height—here we use the DoodleView's width and height as the Bitmap's dimensions. The last argument to createBitmap is the Bitmap's encoding, which specifies how each pixel in the Bitmap is stored. The constant Bitmap.Config.ARGB_8888 indicates that each pixel's color is stored in four bytes (one byte each for the alpha, red, green and blue values) of the pixel's color. Next, we create a new Canvas that's used to draw shapes directly to the Bitmap. Finally, we use Bitmap's eraseColor method to fill the Bitmap with white pixels—the default Bitmap background is black.

DoodleView Methods clear, setDrawingColor, getDrawingColor, setLineWidth and getLineWidth

Figure 7.24 defines methods clear (lines 75–81), setDrawingColor (lines 84–87), getDrawingColor (lines 90–93), setLineWidth (lines 96–99) and getLineWidth (lines 102–105), which are called from the DoodleFragment. Method clear, which we use in the EraseImageDialogFragment, empties the pathMap and previousPointMap, erases the Bitmap by setting all of its pixels to white, then calls the inherited View method **invalidate** to indicate that the View needs to be redrawn. Then, the system automatically determines when the View's onDraw method should be called. Method setDrawingColor changes the current drawing color by setting the color of the Paint object paintLine. Paint's setColor method receives an int that represents the new color in ARGB format. Method getDrawingColor returns the current color, which we use in the ColorDialogFragment. Method setLineWidth sets paintLine's stroke width to the specified number of pixels. Method getLineWidth returns the current stroke width, which we use in the LineWidthDialogFragment.

```
74      // clear the painting
75      public void clear()
76      {
77         pathMap.clear(); // remove all paths
78         previousPointMap.clear(); // remove all previous points
79         bitmap.eraseColor(Color.WHITE); // clear the bitmap
80         invalidate(); // refresh the screen
81      }
82
83      // set the painted line's color
84      public void setDrawingColor(int color)
85      {
86         paintLine.setColor(color);
87      }
88
89      // return the painted line's color
90      public int getDrawingColor()
91      {
92         return paintLine.getColor();
93      }
94
```

Fig. 7.24 | DoodleView methods clear, setDrawingColor, getDrawingColor, setLineWidth and getLineWidth. (Part 1 of 2.)

```
 95     // set the painted line's width
 96     public void setLineWidth(int width)
 97     {
 98        paintLine.setStrokeWidth(width);
 99     }
100
101     // return the painted line's width
102     public int getLineWidth()
103     {
104        return (int) paintLine.getStrokeWidth();
105     }
106
```

Fig. 7.24 | `DoodleView` methods `clear`, `setDrawingColor`, `getDrawingColor`, `setLineWidth` and `getLineWidth`. (Part 2 of 2.)

Overridden `View` *Method* `onDraw`

When a `View` needs to be *redrawn*, its **`onDraw`** method is called. Figure 7.25 overrides `onDraw` to display `bitmap` (the `Bitmap` that contains the drawing) on the `DoodleView` by calling the `Canvas` argument's **`drawBitmap`** method. The first argument is the `Bitmap` to draw, the next two arguments are the *x-y* coordinates where the upper-left corner of the `Bitmap` should be placed on the `View` and the last argument is the `Paint` object that specifies the drawing characteristics. Lines 115–116 then loop through and display the `Paths` that are currently being drawn. For each `Integer` key in the `pathMap`, we pass the corresponding `Path` to `Canvas`'s **`drawPath`** method to draw the `Path` using the `paintLine` object, which defines the line *width* and *color*.

```
107     // called each time this View is drawn
108     @Override
109     protected void onDraw(Canvas canvas)
110     {
111        // draw the background screen
112        canvas.drawBitmap(bitmap, 0, 0, paintScreen);
113
114        // for each path currently being drawn
115        for (Integer key : pathMap.keySet())
116           canvas.drawPath(pathMap.get(key), paintLine); // draw line
117     }
118
```

Fig. 7.25 | Overridden `View` method `onDraw`.

`DoodleView` *Methods* `hideSystemBars` *and* `showSystemBars`

This app uses Android 4.4's new *immersive mode* to allow users to draw on the entire screen. When the user taps the screen, a `GestureDetector`'s `SimplyOnGestureListener` (Fig. 7.27) determines whether the system bars and action bar are displayed. If so, method `hideSystemBars` (Fig. 7.26, lines 120–130) is called; otherwise, method `showSystemBars` (Fig. 7.26, lines 133–140) is called. For this app, we enable immersive mode only for Android 4.4. So, both methods first check whether the version of Android running on the device—`Build.VERSION_SDK_INT`—is greater than or equal to the constant for Android

4.4 (API level 19)—`Build.VERSION_CODES_KITKAT`. If so, both methods use `View` method **`setSystemUiVisibility`** to configure the system bars and action bar. To hide the system bars and action bar and place the UI into immersive mode, you pass to `setSystemUi-Visibility` the constants that are combined via the bitwise OR (`|`) operator in lines 124–129. To show the system bars and action bar, you pass to `setSystemUiVisibility` the constants that are combined in lines 137–139. These combinations of `View` constants ensure that the `DoodleView` is *not* resized each time the system bars and action bar are hidden and redisplayed. Instead, the system bars and action bar *overlay* the `DoodleView`—that is, part of the `DoodleView` is temporarily hidden when the system bars and action bar are on the screen. The constant `View.SYSTEM_UI_FLAG_IMMERSIVE` is new in Android 4.4. For more information on immersive mode, visit:

```
http://developer.android.com/training/system-ui/immersive.html
```

```
119     // hide system bars and action bar
120     public void hideSystemBars()
121     {
122        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT)
123           setSystemUiVisibility(
124              View.SYSTEM_UI_FLAG_LAYOUT_STABLE |
125              View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION |
126              View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN |
127              View.SYSTEM_UI_FLAG_HIDE_NAVIGATION |
128              View.SYSTEM_UI_FLAG_FULLSCREEN |
129              View.SYSTEM_UI_FLAG_IMMERSIVE);
130     }
131
132     // show system bars and action bar
133     public void showSystemBars()
134     {
135        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT)
136           setSystemUiVisibility(
137              View.SYSTEM_UI_FLAG_LAYOUT_STABLE |
138              View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION |
139              View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN);
140     }
141
```

Fig. 7.26 | `DoodleView` methods `hideSystemBars` and `showSystemBars`.

Anonymous Inner Class that Implements Interface `SimpleOnGestureListener`

Figure 7.27 creates the `SimpleOnGestureListener` named `singleTapListener`, which was registered at lines 60–61(Fig. 7.22) with the `GestureDetector`. Recall that `SimpleOn-GestureListener` is an adapter class that implements interfaces `OnGestureListener` and `OnDoubleTapListener`. The methods simply return `false`—indicating that the events were *not handled*. We override only the **`onSingleTap` method** (lines 146–155), which is called when the user taps the screen. We determine whether the system bars and app bar are displayed (lines 149–150) by calling method `View` method **`getSystemUiVisibilty`** and combining its result with the constant `View.SYSTEM_UI_FLAG_HIDE_NAVIGATION`. If the result is `0`, the system bars and app bar are currently displayed, so we call method `hide-`

SystemBars; otherwise, we call showSystemBars. Returning true indicates that the single-tap event has been handled.

```
   // create SimpleOnGestureListener for single tap events
   private SimpleOnGestureListener singleTapListener =
      new SimpleOnGestureListener()
      {
         @Override
         public boolean onSingleTapUp(MotionEvent e)
         {
            if ((getSystemUiVisibility() &
               View.SYSTEM_UI_FLAG_HIDE_NAVIGATION) == 0)
               hideSystemBars();
            else
               showSystemBars();
            return true;
         }
      };

```

Fig. 7.27 | Anonymous inner class that implements interface SimpleOnGestureListener.

Overridden View *Method* onTouchEvent

Method onTouchEvent (Fig. 7.28) is called when the View receives a touch event. Android supports *multitouch*—that is, having multiple fingers touching the screen. At any time, the user can touch the screen with more fingers or remove fingers from the screen. For this reason, each finger—known as a *pointer*—has a unique ID that identifies it across touch events. We'll use that ID to locate the corresponding Path objects that represent each line currently being drawn. These Paths are stored in pathMap.

```
   // handle touch event
   @Override
   public boolean onTouchEvent(MotionEvent event)
   {
      // get the event type and the ID of the pointer that caused the event
      // if a single tap event occurred on KitKat or higher device
      if (singleTapDetector.onTouchEvent(event))
         return true;

      int action = event.getActionMasked(); // event type
      int actionIndex = event.getActionIndex(); // pointer (i.e., finger)

      // determine whether touch started, ended or is moving
      if (action == MotionEvent.ACTION_DOWN ||
         action == MotionEvent.ACTION_POINTER_DOWN)
      {
         touchStarted(event.getX(actionIndex), event.getY(actionIndex),
            event.getPointerId(actionIndex));
      }
```

Fig. 7.28 | Overridden View method onTouchEvent. (Part 1 of 2.)

```
177         else if (action == MotionEvent.ACTION_UP ||
178            action == MotionEvent.ACTION_POINTER_UP)
179         {
180            touchEnded(event.getPointerId(actionIndex));
181         }
182         else
183         {
184            touchMoved(event);
185         }
186
187         invalidate(); // redraw
188         return true;
189      } // end method onTouchEvent
190
```

Fig. 7.28 | Overridden `View` method `onTouchEvent`. (Part 2 of 2.)

When a touch event occurs, line 164 calls the `GestureDetector` (`singleTapDetector`) method `onTouchEvent` to first determine if the touch event was a tap to hide or show the system bars and app bar. If the motion event was a tap, the method returns immediately.

`MotionEvent`'s **`getActionMasked`** method (line 167) returns an `int` representing the `MotionEvent` type, which you can use with constants from class `MotionEvent` to determine how to handle each event. `MotionEvent`'s **`getActionIndex`** method (line 168) returns an integer index representing which finger caused the event. This index is *not* the finger's unique ID—it's simply the index at which that finger's information is located in this `MotionEvent` object. To get the finger's unique ID that persists across `MotionEvent`s until the user removes that finger from the screen, we'll use `MotionEvent`'s **`getPointerID`** method (lines 175 and 180), passing the finger index as an argument.

If the action is `MotionEvent.ACTION_DOWN` or `MotionEvent.ACTION_POINTER_DOWN` (lines 171–172), the user *touched the screen with a new finger*. The first finger to touch the screen generates a `MotionEvent.ACTION_DOWN` event, and all other fingers generate `MotionEvent.ACTION_POINTER_DOWN` events. For these cases, we call the `touchStarted` method (Fig. 7.29) to store the initial coordinates of the touch. If the action is `MotionEvent.ACTION_UP` or `MotionEvent.ACTION_POINTER_UP`, the user *removed a finger from the screen*, so we call method `touchEnded` (Fig. 7.31) to draw the completed `Path` to the `bitmap` so that we have a permanent record of that `Path`. For all other touch events, we call method `touchMoved` (Fig. 7.30) to draw the lines. After the event is processed, line 187 calls the inherited `View` method `invalidate` to redraw the screen, and line 188 returns `true` to indicate that the event has been processed.

`touchStarted` Method of Class `DoodleView`

The `touchStarted` method (Fig. 7.29) is called when a finger first *touches* the screen. The coordinates of the touch and its ID are supplied as arguments. If a `Path` already exists for the given ID (line 198), we call `Path`'s **`reset`** method to *clear* any existing points so we can *reuse* the `Path` for a new stroke. Otherwise, we create a new `Path`, add it to `pathMap`, then add a new `Point` to the `previousPointMap`. Lines 213–215 call `Path`'s **`moveTo`** method to set the `Path`'s starting coordinates and specify the new `Point`'s `x` and `y` values.

```java
   // called when the user touches the screen
   private void touchStarted(float x, float y, int lineID)
   {
      Path path; // used to store the path for the given touch id
      Point point; // used to store the last point in path

      // if there is already a path for lineID
      if (pathMap.containsKey(lineID))
      {
         path = pathMap.get(lineID); // get the Path
         path.reset(); // reset the Path because a new touch has started
         point = previousPointMap.get(lineID); // get Path's last point
      }
      else
      {
         path = new Path();
         pathMap.put(lineID, path); // add the Path to Map
         point = new Point(); // create a new Point
         previousPointMap.put(lineID, point); // add the Point to the Map
      }

      // move to the coordinates of the touch
      path.moveTo(x, y);
      point.x = (int) x;
      point.y = (int) y;
   } // end method touchStarted

```

Fig. 7.29 | `touchStarted` method of class `DoodleView`.

`touchMoved` Method of Class `DoodleView`

The `touchMoved` method (Fig. 7.30) is called when the user moves one or more fingers across the screen. The system `MotionEvent` passed from `onTouchEvent` contains touch information for multiple moves on the screen if they occur at the same time. `MotionEvent` method **`getPointerCount`** (line 222) returns the number of touches this `MotionEvent` describes. For each, we store the finger's ID (line 225) in `pointerID`, and store the finger's corresponding index in this `MotionEvent` (line 226) in `pointerIndex`. Then we check whether there's a corresponding `Path` in the `pathMap` `HashMap` (line 229). If so, we use `MotionEvent`'s `getX` and `getY` methods to get the last coordinates for this *drag* event for the specified `pointerIndex`. We get the corresponding `Path` and last `Point` for the `pointerID` from each respective `HashMap`, then calculate the difference between the last point and the current point—we want to update the `Path` *only* if the user has moved a distance that's greater than our `TOUCH_TOLERANCE` constant. We do this because many devices are sensitive enough to generate `MotionEvent`s indicating small movements when the user is attempting to hold a finger motionless on the screen. If the user moved a finger further than the `TOUCH_TOLERANCE`, we use `Path`'s **`quadTo`** method (lines 248–249) to add a geometric curve (specifically a *quadratic Bezier curve*) from the previous `Point` to the new `Point`. We then update the most recent `Point` for that finger.

```java
   // called when the user drags along the screen
   private void touchMoved(MotionEvent event)
   {
      // for each of the pointers in the given MotionEvent
      for (int i = 0; i < event.getPointerCount(); i++)
      {
         // get the pointer ID and pointer index
         int pointerID = event.getPointerId(i);
         int pointerIndex = event.findPointerIndex(pointerID);

         // if there is a path associated with the pointer
         if (pathMap.containsKey(pointerID))
         {
            // get the new coordinates for the pointer
            float newX = event.getX(pointerIndex);
            float newY = event.getY(pointerIndex);

            // get the Path and previous Point associated with
            // this pointer
            Path path = pathMap.get(pointerID);
            Point point = previousPointMap.get(pointerID);

            // calculate how far the user moved from the last update
            float deltaX = Math.abs(newX - point.x);
            float deltaY = Math.abs(newY - point.y);

            // if the distance is significant enough to matter
            if (deltaX >= TOUCH_TOLERANCE || deltaY >= TOUCH_TOLERANCE)
            {
               // move the path to the new location
               path.quadTo(point.x, point.y, (newX + point.x) / 2,
                  (newY + point.y) / 2);

               // store the new coordinates
               point.x = (int) newX;
               point.y = (int) newY;
            }
         }
      }
   } // end method touchMoved

```

Fig. 7.30 | `touchMoved` method of class `DoodleView`.

`touchEnded` Method of Class `DoodleView`

The `touchEnded` method (Fig. 7.31) is called when the user lifts a finger from the screen. The method receives the ID of the finger (`lineID`) for which the touch just ended as an argument. Line 262 gets the corresponding `Path`. Line 263 calls the `bitmapCanvas`'s `drawPath` method to draw the `Path` on the `Bitmap` object named `bitmap` before we call `Path`'s `reset` method to clear the `Path`. Resetting the `Path` does not erase its corresponding painted line from the screen, because those lines have already been drawn to the `bitmap` that's displayed to the screen. The lines that are currently being drawn by the user are displayed on top of that `bitmap`.

```java
   // called when the user finishes a touch
   private void touchEnded(int lineID)
   {
      Path path = pathMap.get(lineID); // get the corresponding Path
      bitmapCanvas.drawPath(path, paintLine); // draw to bitmapCanvas
      path.reset(); // reset the Path
   }

```

Fig. 7.31 | `touchEnded` method of class `DoodleView`.

`DoodleView` Method `saveImage`

Method `saveImage` (Fig. 7.32) saves the current drawing to a file in the device's gallery. Line 271 creates a filename for the image, then lines 274–276 store the image in the device's **Gallery** by calling class `MediaStore.Images.Media`'s `insertImage` method. The method receives four arguments:

- a `ContentResolver` that the method uses to locate where the image should be stored on the device
- the `Bitmap` to store
- the name of the image
- a description of the image

Method `insertImage` returns a `String` representing the image's location on the device, or null if the image could not be saved. Lines 278–295 check whether the image was saved and display an appropriate `Toast`.

```java
267    // save the current image to the Gallery
268    public void saveImage()
269    {
270       // use "Doodlz" followed by current time as the image name
271       String name = "Doodlz" + System.currentTimeMillis() + ".jpg";
272
273       // insert the image in the device's gallery
274       String location = MediaStore.Images.Media.insertImage(
275          getContext().getContentResolver(), bitmap, name,
276          "Doodlz Drawing");
277
278       if (location != null) // image was saved
279       {
280          // display a message indicating that the image was saved
281          Toast message = Toast.makeText(getContext(),
282             R.string.message_saved, Toast.LENGTH_SHORT);
283          message.setGravity(Gravity.CENTER, message.getXOffset() / 2,
284             message.getYOffset() / 2);
285          message.show();
286       }
287       else
288       {
```

Fig. 7.32 | `DoodleView` method `saveImage`. (Part 1 of 2.)

```
            // display a message indicating that the image was saved
            Toast message = Toast.makeText(getContext(),
               R.string.message_error_saving, Toast.LENGTH_SHORT);
            message.setGravity(Gravity.CENTER, message.getXOffset() / 2,
               message.getYOffset() / 2);
            message.show();
         }
      } // end method saveImage

```

Fig. 7.32 | DoodleView method saveImage. (Part 2 of 2.)

DoodleView Method printImage

On Android 4.4 and higher devices, method printImage (Fig. 7.33) uses the Android Support Library's PrintHelper class to print the current drawing. Line 301 first confirms that printing support is available on the device. If so, line 304 creates a PrintHelper object. Next, line 307 specifies the image's *scale mode*—PrintHelper.SCALE_MODE_FIT indicates that the image should fit within the printable area of the paper. There's also the scale mode PrintHelper.SCALE_MODE_FILL, which causes the image to fill the paper, possibly cutting off a portion of the image. Finally, line 308 calls PrintHelper method **printBitmap**, passing as arguments the print job name (used by the printer to identify the print) and the Bitmap containing the image to print. This displays Android's print dialog, which allows the user to choose whether to save the image as a PDF document on the device or to print the image to an available printer.

```
298      // print the current image
299      public void printImage()
300      {
301         if (PrintHelper.systemSupportsPrint())
302         {
303            // use Android Support Library's PrintHelper to print image
304            PrintHelper printHelper = new PrintHelper(getContext());
305
306            // fit image in page bounds and print the image
307            printHelper.setScaleMode(PrintHelper.SCALE_MODE_FIT);
308            printHelper.printBitmap("Doodlz Image", bitmap);
309         }
310         else
311         {
312            // display message indicating that system does not allow printing
313            Toast message = Toast.makeText(getContext(),
314               R.string.message_error_printing, Toast.LENGTH_SHORT);
315            message.setGravity(Gravity.CENTER, message.getXOffset() / 2,
316               message.getYOffset() / 2);
317            message.show();
318         }
319      }
320   } // end class DoodleView
```

Fig. 7.33 | DoodleView method printImage.

7.7 ColorDialogFragment Class

Class ColorDialogFragment (Figs. 7.34–7.38) extends DialogFragment to create an AlertDialog for setting the drawing color. The class's instance variables (lines 19–24) are used to reference the GUI controls for selecting the new color, displaying a preview of it and storing the color as a 32-bit int value that represents the color's ARGB values.

```
 1  // ColorDialogFragment.java
 2  // Allows user to set the drawing color on the DoodleView
 3  package com.deitel.doodlz;
 4
 5  import android.app.Activity;
 6  import android.app.AlertDialog;
 7  import android.app.Dialog;
 8  import android.app.DialogFragment;
 9  import android.content.DialogInterface;
10  import android.graphics.Color;
11  import android.os.Bundle;
12  import android.view.View;
13  import android.widget.SeekBar;
14  import android.widget.SeekBar.OnSeekBarChangeListener;
15
16  // class for the Select Color dialog
17  public class ColorDialogFragment extends DialogFragment
18  {
19     private SeekBar alphaSeekBar;
20     private SeekBar redSeekBar;
21     private SeekBar greenSeekBar;
22     private SeekBar blueSeekBar;
23     private View colorView;
24     private int color;
25
```

Fig. 7.34 | ColorDialogFragment's package statement, import statements and instance variables.

Overridden ***DialogFragment*** *Method* ***onCreateDialog***

Method onCreateDialog (Fig. 7.35) inflates the custom View (lines 32–34) defined by fragment_color.xml containing the GUI for selecting a color, then attaches that View to the AlertDialog by calling AlertDialog.Builder's setView method (line 35). Lines 42–50 get references to the dialog's SeekBars and colorView. Next, lines 53–56 register colorChangedListener (Fig. 7.38) as the listener for the SeekBars' events.

```
26     // create an AlertDialog and return it
27     @Override
28     public Dialog onCreateDialog(Bundle bundle)
29     {
30        AlertDialog.Builder builder =
31           new AlertDialog.Builder(getActivity());
```

Fig. 7.35 | Overridden DialogFragment method onCreateDialog. (Part 1 of 2.)

```
32       View colorDialogView =
33          getActivity().getLayoutInflater().inflate(
34             R.layout.fragment_color, null);
35       builder.setView(colorDialogView); // add GUI to dialog
36
37       // set the AlertDialog's message
38       builder.setTitle(R.string.title_color_dialog);
39       builder.setCancelable(true);
40
41       // get the color SeekBars and set their onChange listeners
42       alphaSeekBar = (SeekBar) colorDialogView.findViewById(
43          R.id.alphaSeekBar);
44       redSeekBar = (SeekBar) colorDialogView.findViewById(
45          R.id.redSeekBar);
46       greenSeekBar = (SeekBar) colorDialogView.findViewById(
47          R.id.greenSeekBar);
48       blueSeekBar = (SeekBar) colorDialogView.findViewById(
49          R.id.blueSeekBar);
50       colorView = colorDialogView.findViewById(R.id.colorView);
51
52       // register SeekBar event listeners
53       alphaSeekBar.setOnSeekBarChangeListener(colorChangedListener);
54       redSeekBar.setOnSeekBarChangeListener(colorChangedListener);
55       greenSeekBar.setOnSeekBarChangeListener(colorChangedListener);
56       blueSeekBar.setOnSeekBarChangeListener(colorChangedListener);
57
58       // use current drawing color to set SeekBar values
59       final DoodleView doodleView = getDoodleFragment().getDoodleView();
60       color = doodleView.getDrawingColor();
61       alphaSeekBar.setProgress(Color.alpha(color));
62       redSeekBar.setProgress(Color.red(color));
63       greenSeekBar.setProgress(Color.green(color));
64       blueSeekBar.setProgress(Color.blue(color));
65
66       // add Set Color Button
67       builder.setPositiveButton(R.string.button_set_color,
68          new DialogInterface.OnClickListener()
69          {
70             public void onClick(DialogInterface dialog, int id)
71             {
72                doodleView.setDrawingColor(color);
73             }
74          }
75       ); // end call to setPositiveButton
76
77       return builder.create(); // return dialog
78    } // end method onCreateDialog
79
```

Fig. 7.35 | Overridden `DialogFragment` method `onCreateDialog`. (Part 2 of 2.)

Line 59 calls method `getDoodleFragment` (Fig. 7.36) to get a reference to the `DoodleFragment`, then calls the `DoodleFragment`'s `getDoodleView` method to get the `DoodleView`. Lines 60–64 get the `DoodleView`'s current drawing color, then use it to set each

SeekBar's value. Color's static methods **alpha**, **red**, **green** and **blue** extract the ARGB values from the color, and SeekBar's setProgress method positions the thumbs. Lines 67–75 configure the AlertDialog's positive button to set the DoodleView's new drawing color. Line 77 returns the AlertDialog.

Method **getDoodleFragment**

Method getDoodleFragment (Fig. 7.36) simply uses the FragmentManager to get a reference to the DoodleFragment.

```
   // gets a reference to the DoodleFragment
   private DoodleFragment getDoodleFragment()
   {
      return (DoodleFragment) getFragmentManager().findFragmentById(
         R.id.doodleFragment);
   }

```

Fig. 7.36 | Method getDoodleFragment.

Overridden **Fragment** *Lifecycle Methods* **onAttach** *and* **onDetach**

When the ColorDialogFragment is added to a parent Activity, method onAttach (Fig. 7.37, lines 88–96) is called. Line 92 gets a reference to the DoodleFragment. If that reference is not null, line 95 calls DoodleFragment's setDialogOnScreen method to indicate that the **Choose Color** dialog is now displayed. When the ColorDialogFragment is removed from a parent Activity, method onDetach (lines 99–107) is called. Line 106 calls DoodleFragment's setDialogOnScreen method to indicate that the **Choose Color** dialog is no longer on the screen.

```
87     // tell DoodleFragment that dialog is now displayed
88     @Override
89     public void onAttach(Activity activity)
90     {
91        super.onAttach(activity);
92        DoodleFragment fragment = getDoodleFragment();
93
94        if (fragment != null)
95           fragment.setDialogOnScreen(true);
96     }
97
98     // tell DoodleFragment that dialog is no longer displayed
99     @Override
100    public void onDetach()
101    {
102       super.onDetach();
103       DoodleFragment fragment = getDoodleFragment();
104
105       if (fragment != null)
106          fragment.setDialogOnScreen(false);
107    }
108
```

Fig. 7.37 | Overridden Fragment lifecycle methods onAttach and onDetach.

Anonymous Inner Class That Implements Interface **OnSeekBarChangeListener** *to Respond to the Events of the Alpha, Red, Green and Blue* **SeekBars**

Figure 7.38 defines an anonymous inner class that implements interface OnSeekBar-ChangeListener to respond to events when the user adjusts the SeekBars in the **Choose Color** Dialog. This was registered as the SeekBars' event handler in Fig. 7.35 (lines 53–56). Method onProgressChanged (lines 115–123) is called when the position of a SeekBar's thumb changes. If the user moved a SeekBar's thumb (line 118), lines 119–121 store the new color. Class Color's static method **argb** combines the SeekBars' values into a Color and returns the appropriate color as an int. We then use class View's **setBack-groundColor** method to update the colorView with a color that matches the current state of the SeekBars.

```
109     // OnSeekBarChangeListener for the SeekBars in the color dialog
110     private OnSeekBarChangeListener colorChangedListener =
111       new OnSeekBarChangeListener()
112     {
113        // display the updated color
114        @Override
115        public void onProgressChanged(SeekBar seekBar, int progress,
116           boolean fromUser)
117        {
118           if (fromUser) // user, not program, changed SeekBar progress
119              color = Color.argb(alphaSeekBar.getProgress(),
120                 redSeekBar.getProgress(), greenSeekBar.getProgress(),
121                 blueSeekBar.getProgress());
122           colorView.setBackgroundColor(color);
123        }
124
125        @Override
126        public void onStartTrackingTouch(SeekBar seekBar) // required
127        {
128        }
129
130        @Override
131        public void onStopTrackingTouch(SeekBar seekBar) // required
132        {
133        }
134     }; // end colorChanged
135  } // end class ColorDialogFragment
```

Fig. 7.38 | Anonymous inner class that implements interface OnSeekBarChangeListener to respond to the events of the alpha, red, green and blue SeekBars.

7.8 LineWidthDialogFragment Class

Class LineWidthDialogFragment (Fig. 7.39) extends DialogFragment to create an Alert-Dialog for setting the line width. The class is similar to class ColorDialogFragment, so we discuss only the key differences here. The class's only instance variable is an ImageView (line 22) in which we draw a line showing the current line-width setting.

```
// LineWidthDialogFragment.java
// Allows user to set the drawing color on the DoodleView
package com.deitel.doodlz;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.content.DialogInterface;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.os.Bundle;
import android.view.View;
import android.widget.ImageView;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;

// class for the Select Color dialog
public class LineWidthDialogFragment extends DialogFragment
{
   private ImageView widthImageView;

   // create an AlertDialog and return it
   @Override
   public Dialog onCreateDialog(Bundle bundle)
   {
      AlertDialog.Builder builder =
         new AlertDialog.Builder(getActivity());
      View lineWidthDialogView = getActivity().getLayoutInflater().inflate(
         R.layout.fragment_line_width, null);
      builder.setView(lineWidthDialogView); // add GUI to dialog

      // set the AlertDialog's message
      builder.setTitle(R.string.title_line_width_dialog);
      builder.setCancelable(true);

      // get the ImageView
      widthImageView = (ImageView) lineWidthDialogView.findViewById(
         R.id.widthImageView);

      // configure widthSeekBar
      final DoodleView doodleView = getDoodleFragment().getDoodleView();
      final SeekBar widthSeekBar = (SeekBar)
         lineWidthDialogView.findViewById(R.id.widthSeekBar);
      widthSeekBar.setOnSeekBarChangeListener(lineWidthChanged);
      widthSeekBar.setProgress(doodleView.getLineWidth());

      // add Set Line Width Button
      builder.setPositiveButton(R.string.button_set_line_width,
         new DialogInterface.OnClickListener()
         {
```

Fig. 7.39 | Class `LineWidthDialogFragment`. (Part 1 of 3.)

```
            public void onClick(DialogInterface dialog, int id)
            {
               doodleView.setLineWidth(widthSeekBar.getProgress());
            }
         }
      ); // end call to setPositiveButton

      return builder.create(); // return dialog
   } // end method onCreateDialog

   // gets a reference to the DoodleFragment
   private DoodleFragment getDoodleFragment()
   {
      return (DoodleFragment) getFragmentManager().findFragmentById(
         R.id.doodleFragment);
   }

   // tell DoodleFragment that dialog is now displayed
   @Override
   public void onAttach(Activity activity)
   {
      super.onAttach(activity);
      DoodleFragment fragment = getDoodleFragment();

      if (fragment != null)
         fragment.setDialogOnScreen(true);
   }

   // tell DoodleFragment that dialog is no longer displayed
   @Override
   public void onDetach()
   {
      super.onDetach();
      DoodleFragment fragment = getDoodleFragment();

      if (fragment != null)
         fragment.setDialogOnScreen(false);
   }

   // OnSeekBarChangeListener for the SeekBar in the width dialog
   private OnSeekBarChangeListener lineWidthChanged =
      new OnSeekBarChangeListener()
      {
         Bitmap bitmap = Bitmap.createBitmap(
            400, 100, Bitmap.Config.ARGB_8888);
         Canvas canvas = new Canvas(bitmap); // associate with Canvas

         @Override
         public void onProgressChanged(SeekBar seekBar, int progress,
            boolean fromUser)
         {
```

Fig. 7.39 | Class `LineWidthDialogFragment`. (Part 2 of 3.)

```
104             // configure a Paint object for the current SeekBar value
105             Paint p = new Paint();
106             p.setColor(
107                getDoodleFragment().getDoodleView().getDrawingColor());
108             p.setStrokeCap(Paint.Cap.ROUND);
109             p.setStrokeWidth(progress);
110
111             // erase the bitmap and redraw the line
112             bitmap.eraseColor(
113                getResources().getColor(android.R.color.transparent));
114             canvas.drawLine(30, 50, 370, 50, p);
115             widthImageView.setImageBitmap(bitmap);
116          }
117
118          @Override
119          public void onStartTrackingTouch(SeekBar seekBar) // required
120          {
121          }
122
123          @Override
124          public void onStopTrackingTouch(SeekBar seekBar)  // required
125          {
126          }
127       }; // end lineWidthChanged
128 }
```

Fig. 7.39 | Class `LineWidthDialogFragment`. (Part 3 of 3.)

Method `onCreateDialog`

Method `onCreateDialog` (lines 25–61) inflates the custom `View` (lines 30–31) defined by `fragment_line_width.xml` that displays the GUI for selecting the line width, then attaches that `View` to the `AlertDialog` by calling `AlertDialog.Builder`'s `setView` method (line 32). Lines 39–40 get a reference to the `ImageView` in which the sample line will be drawn. Next, lines 43–47 get a reference to the `widthSeekBar`, register `lineWidthChanged` (lines 93–127) as the `SeekBar`'s listener and set the `SeekBar`'s current value to the current line width. Lines 50–58 define the dialog's positive button to call the `DoodleView`'s `setLineWidth` method when the user touches the **Set Line Width** button. Line 60 returns the `AlertDialog` for display.

Anonymous Inner Class That Implements Interface `OnSeekBarChangeListener` *to Respond to the Events of the* `widthSeekBar`

Lines 93–127 define the `lineWidthChanged` `OnSeekBarChangeListener` that responds to events when the user adjusts the `SeekBar` in the **Choose Line Width** dialog. Lines 96–97 create a `Bitmap` on which to display a sample line representing the selected line thickness. Line 98 creates a `Canvas` for drawing on the `Bitmap`. Method `onProgressChanged` (lines 100–116) draws the sample line based on the current drawing color and the `SeekBar`'s value. First, lines 105–109 configure a `Paint` object for drawing the sample line. Class `Paint`'s **`setStrokeCap`** method (line 108) specifies the appearance of the line ends—in this case, they're rounded (`Paint.Cap.ROUND`). Lines 112–113 clear `bitmap`'s background to the predefined Android color `android.R.color.transparent` with `Bitmap` method

eraseColor. We use canvas to draw the sample line. Finally, line 115 displays bitmap in the widthImageView by passing it to ImageView's **setImageBitmap** method.

7.9 EraseImageDialogFragment Class

Class EraseImageDialogFragment (Fig. 7.40) extends DialogFragment to create an AlertDialog that confirms whether the user really wants to erase the entire image. The class is similar to class ColorDialogFragment and LineWidthDialogFragment, so we discuss only method onCreateDialog (lines 16–41) here. The method creates an AlertDialog with **Erase Image** and **Cancel** button. Lines 27–35 configure the **Erase Image** button as the positive button—when the user touches this, line 32 in the button's listener calls the DoodleView's clear method to erase the image. Line 38 configures **Cancel** as the negative button—when the user touches this, the dialog is dismissed. Line 40 returns the AlertDialog.

```
 1  // EraseImageDialogFragment.java
 2  // Allows user to erase image
 3  package com.deitel.doodlz;
 4
 5  import android.app.Activity;
 6  import android.app.AlertDialog;
 7  import android.app.Dialog;
 8  import android.app.DialogFragment;
 9  import android.content.DialogInterface;
10  import android.os.Bundle;
11
12  //class for the Select Color dialog
13  public class EraseImageDialogFragment extends DialogFragment
14  {
15     // create an AlertDialog and return it
16     @Override
17     public Dialog onCreateDialog(Bundle bundle)
18     {
19        AlertDialog.Builder builder =
20           new AlertDialog.Builder(getActivity());
21
22        // set the AlertDialog's message
23        builder.setMessage(R.string.message_erase);
24        builder.setCancelable(false);
25
26        // add Erase Button
27        builder.setPositiveButton(R.string.button_erase,
28           new DialogInterface.OnClickListener()
29           {
30              public void onClick(DialogInterface dialog, int id)
31              {
32                 getDoodleFragment().getDoodleView().clear(); // clear image
33              }
34           }
35        ); // end call to setPositiveButton
```

Fig. 7.40 | Class EraseImageDialogFragment. (Part 1 of 2.)

```

      // add Cancel Button
      builder.setNegativeButton(R.string.button_cancel, null);

      return builder.create(); // return dialog
   } // end method onCreateDialog

   // gets a reference to the DoodleFragment
   private DoodleFragment getDoodleFragment()
   {
      return (DoodleFragment) getFragmentManager().findFragmentById(
         R.id.doodleFragment);
   }

   // tell DoodleFragment that dialog is now displayed
   @Override
   public void onAttach(Activity activity)
   {
      super.onAttach(activity);
      DoodleFragment fragment = getDoodleFragment();

      if (fragment != null)
         fragment.setDialogOnScreen(true);
   }

   // tell DoodleFragment that dialog is no longer displayed
   @Override
   public void onDetach()
   {
      super.onDetach();
      DoodleFragment fragment = getDoodleFragment();

      if (fragment != null)
         fragment.setDialogOnScreen(false);
   }
} // end class EraseImageDialogFragment
```

Fig. 7.40 | Class `EraseImageDialogFragment`. (Part 2 of 2.)

7.10 Wrap-Up

In this chapter, you built the **Doodlz** app which enables users to paint by dragging one or more fingers across the screen. You implemented a shake-to-erase feature by using Android's `SensorManager` to register a `SensorEventListener` that responds to accelerometer events, and you learned that Android supports many other sensors.

You created subclasses of `DialogFragment` that displayed custom `View`s in `AlertDialog`s. You also overrode the `Fragment` lifecycle methods `onAttach` and `onDetach`, which are called when a `Fragment` is attached to or detached from a parent `Activity`, respectively.

We showed how to associate a `Canvas` with a `Bitmap`, then use the `Canvas` to draw on the `Bitmap`. We demonstrated how to handle multitouch events so the user can draw with multiple fingers at the same time. You stored the information for each individual finger as a `Path`. You processed the touch events by overriding the `View` method `onTouchEvent`, which receives a `MotionEvent` containing the event type and the ID of the pointer that generated the event. We used the IDs to distinguish among the fingers and add information to the corresponding `Path` objects.

You used Android 4.4's new full-screen immersive mode that enables an app to take advantage of the entire screen, but still allows the user to access the system bars and action bar when necessary. To toggle immersive mode, you used a `GestureDetector` to determine when the user single-tapped the screen.

You used a `ContentResolver` and the `MediaStore.Images.Media.insertImage` method to save an image into the device's **Gallery**. Finally, we showed how to use the new Android 4.4 printing framework to allow users to print their drawings. You used the Android Support Library's `PrintHelper` class to print a `Bitmap`Bitmap. The `PrintHelper` displayed a user interface for selecting a printer or saving the image into a PDF document.

In Chapter 8, we build the database-driven **Address Book** app, which provides quick and easy access to stored contact information and the ability to add contacts, delete contacts and edit existing contacts. You'll learn how to dynamically swap `Fragments` in a GUI and once again provide layouts that optimize screen real estate on phones and tablets.

Self-Review Exercises

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

a) You use the `SensorManager` to register the sensor changes that your app should receive and to specify the ________ that will handle those sensor-change events.

b) A `Path` object (package `android.graphics`) represents a geometric path consisting of line segments and ________.

c) You use the type of the touch event to determine whether the user has touched the screen, ________ or lifted a finger from the screen.

d) Use class `SensorManager`'s ________ method to stop listening for accelerometer events.

e) Override `SensorEventListener` method ________ to process accelerometer events.

f) Override `Fragment` method ________ to respond to the event when a Fragment is attached to a parent `Activity`.

g) When a `View` needs to be redrawn, its ________ method is called.

h) `MotionEvent`'s ________ method returns an `int` representing the `MotionEvent` type, which you can use with constants from class `MotionEvent` to determine how to handle each event.

i) Android 4.4's ________ enables an app to take advantage of the entire screen.

7.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) You unregister the accelerometer event handler when the app is sent to the foreground.

b) Call the inherited `View` method `validate` to indicate that the `View` needs to be redrawn.

c) If the action is `MotionEvent.ACTION_DOWN` or `MotionEvent.ACTION_POINTER_DOWN`, the user touched the screen with the same finger.

d) Resetting the `Path` erases its corresponding painted line from the screen, because those lines have already been drawn to the `bitmap` that's displayed to the screen.

e) Method `MediaStore.Images.Media.saveImage` saves a `Bitmap` into the device's **Gallery**.

Answers to Self-Review Exercises

7.1 a) `SensorEventListener`. b) curves. c) dragged across the screen. d) `unregisterListener`. e) `onSensorChanged`. f) `onAttach`. g) `onDraw`. h) `getActionMasked`. i) immersive mode.

7.2 a) False. You unregister the accelerometer event handler when the app is sent to the *background*. b) False. Call the inherited `View` method `invalidate` to indicate that the `View` needs to be redrawn. c) False. If the action is `MotionEvent.ACTION_DOWN` or `MotionEvent.ACTION_POINTER_DOWN`, the user touched the screen with a new finger. d) False. Resetting the `Path` *does not erase* its corresponding painted line from the screen, because those lines have already been drawn to the `bitmap` that's displayed to the screen. e) False. The method `MediaStore.Images.Media.insertImage` saves a `Bitmap` into the device's **Gallery**.

Exercises

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

a) Most Android devices have a(n) _________ that allows apps to detect movement.
b) A _________ displays an `AlertDialog` with a custom `View` containing GUI components for previewing and selecting a new ARGB drawing color.
c) The _________ monitors the accelerometer to detect device movement.
d) `SensorManager`'s _________ constant represents the acceleration due to gravity on earth.
e) You register to receive accelerometer events using `SensorManager`'s `registerListener` method, which receives three arguments: the `SensorEventListener` object that will respond to the events, a `Sensor` representing the type of sensor data the app wishes to receive and _________.
f) _____________ is the last `Fragment` lifecycle method called when a `Fragment` is about to be detached from a parent `Activity`.
g) `Paint` method _________ sets the stroke width to the specified number of pixels.
h) Android supports _________—that is, having multiple fingers touching the screen.
i) A _________ class of package `android.content` enables the app to read data from and store data on a device.

7.4 State whether each of the following is *true* or *false*. If *false*, explain why.

a) In Android, sensor events are handled in the GUI thread.
b) `EraseImageDialogFragment` displays a standard `AlertDialog` asking the user to confirm whether the entire image should be erased.
c) For accelerometer events, the `SensorEvent` parameter `values` array contains three elements representing the acceleration (in meters/second^2) in the x (left/right), y (up/down) and z (forward/backward) directions.
d) Method `onProgressChanged` is called once when the user drags a `SeekBar`'s thumb.
e) An accelerometer allows an app to react to user interactions such as *flings*, *single taps*, *double taps*, *long presses* and *scrolls*.
f) The system `MotionEvent` passed from `onTouchEvent` contains touch information for multiple moves on the screen if they occur at the same time.

7.5 *(Enhanced* **Doodlz** *App)* Make the following enhancements to the **Doodlz** app:

a) Allow the user to select a background color. The erase capability should use the selected background color. Clearing the entire image should return the background to the default white background.
b) Allow the user to select a background image on which to draw. Clearing the entire image should return the background to the default white background. The erase capability should use the default white background color.

c) Use pressure to determine transparency of color or thickness of line. Class `MotionEvent` has methods that allow you to get the pressure of the touch.
d) Add the ability to draw rectangles and ovals. Options should include whether the shape is filled or hollow. The user should be able to specify the line thickness for each shape's border and the shape's fill color.
e) *(Advanced)* When the user selects a background image on which to draw, the erase capability should reveal the original background image pixels in the erased location.

7.6 *(Hangman Game App)* Recreate the classic word game Hangman using the Android robot icon rather than a stick figure. (For the Android logo terms of use, visit `www.android.com/branding.html`). At the start of the game, display a dashed line with one dash representing each letter in the word. As a hint to the user, provide either a category for the word (e.g., sport or landmark) or the word's definition. Ask the user to enter a letter. If the letter is in the word, place it in the location of the corresponding dash. If the letter is not part of the word, draw part of the Android robot on the screen (e.g., the robot's head). For each incorrect answer, draw another part of the Android robot. The game ends when the user completes the word or the entire Android Robot is drawn to the screen.

7.7 *(Fortune Teller App)* The user "asks a question" then shakes the phone to find a fortune (e.g., "probably not," "looks promising," "ask me again later." etc.

7.8 *(Block Breaker Game)* Display several columns of blocks in red, yellow, blue and green. Each column should have blocks of each color randomly placed. Blocks can be removed from the screen only if they are in groups of two or more. A group consists of blocks of the same color that are vertically and/or horizontally adjacent. When the user taps a group of blocks, the group disappears and the blocks above move down to fill the space. The goal is to clear all of the blocks from the screen. More points should be awarded for larger groups of blocks.

7.9 *(Enhanced Block Breaker Game)* Modify the Block Breaker game in Exercise 7.8 as follows:
a) Provide a timer—the user wins by clearing the blocks in the alotted time. Add more blocks to the screen the longer it takes the user to clear the screen.
b) Add multiple levels. In each level, the alotted time for clearing the screen decreases.
c) Provide a continous mode in which as the user clears blocks, a new row of blocks is added. If the space below a given block is empty, the block should drop into that space. In this mode, the game ends when the user cannot remove any more blocks.
d) Keep track of the high scores in each game mode.

7.10 *(Word Search App)* Create a grid of letters that fills the screen. Hidden in the grid should be at least ten words. The words may be horizontal, vertical or diagonal, and, in each case, forwards, backwards, up or down. Allow the user to highlight the words by dragging a finger across the letters on the screen or tapping each letter of the word. Include a timer. The less time it takes the user to complete the game, the higher the score. Keep track of the high scores.

7.11 *(Fractal App)* Research how to draw fractals and develop an app that draws them. Provide options that allow the user to control the number of levels of the fractal and its colors.

7.12 *(Kaleidascope App)* Create an app that simulates a kaleidascope. Allow the user to shake the device to redraw the screen.

7.13 *(Labyrinth Game App: Open Source)* Check out the open-source Android app, *Amazed*, on the Google Code site (`http://apps-for-android.googlecode.com/svn/trunk/Amazed/`). In this game, the user maneuvers a marble through a maze by tilting the device in various directions. Possible modifications and enhancements include: adding a timer to keep track of how fast the user completes the game, improving the graphics, adding sounds and adding more puzzles of varying difficulty.

7.14 *(Game of Snake App)* Research the Game of Snake online and develop an app that allows a user to play the game.

8 Address Book App

Objectives

In this chapter you'll:

- Use a `ListFragment` to display and manage a `ListView`.
- Use `FragmentTransactions` and the back stack to dynamically attach `Fragments` to and detach `Fragments` from the GUI.
- Create and open SQLite databases using a `SQLiteOpenHelper`, and insert, delete and query data in a SQLite database using a `SQLiteDatabase` object
- Use a `SimpleCursorAdapter` to bind database query results to a `ListView`'s items.
- Use a `Cursor` to manipulate database query results.
- Use multithreading and `AsyncTasks` to perform database operations outside the GUI thread and maintain application responsiveness.
- Define styles containing common GUI attributes and values, then apply them to multiple GUI components.

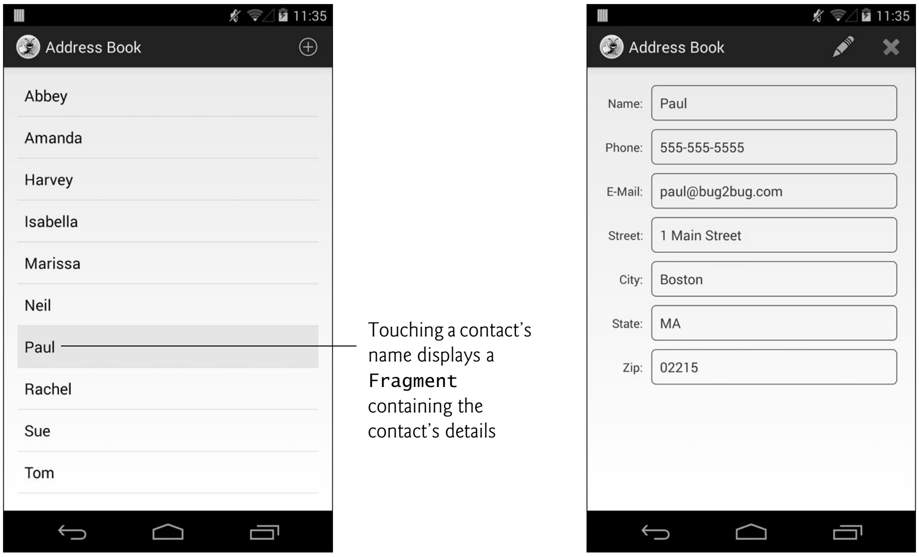
Outline

8.1 Introduction

The **Address Book** app (Fig. 8.1) provides convenient access to contact information that's stored in a SQLite database on the device. You can scroll through an alphabetical contact list and can view a contact's details by touching the contact's name.

a) Contact list with **Paul** selected

b) Details for the contact **Paul**



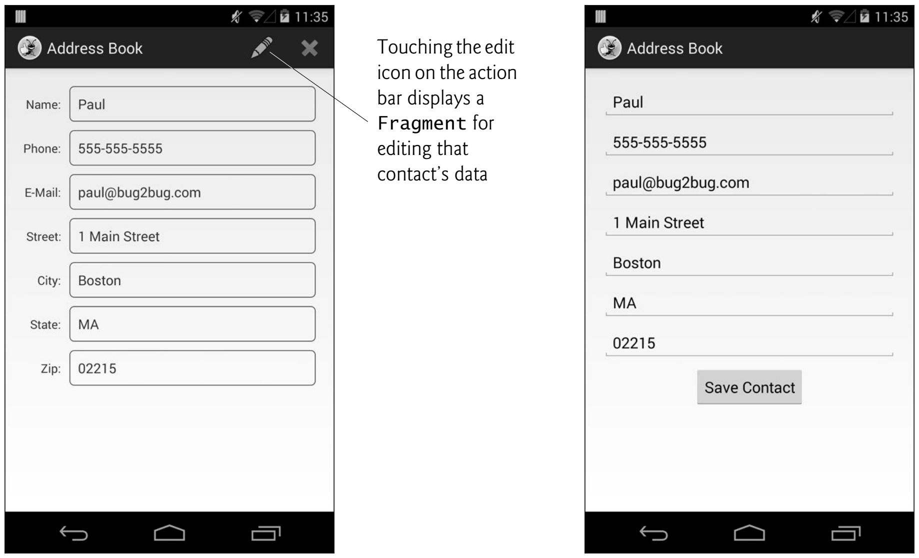
Name: Paul
Phone: 555-555-5555
E-Mail: paul@bug2bug.com
Street: 1 Main Street
City: Boston
State: MA
Zip: 02215

Fig. 8.1 | contact list and a selected contact's details.

When a contact's details are displayed, touching *edit* (✎) displays a `Fragment` containing prepopulated `EditText`s for editing the contact's data (Fig. 8.2), and touching *delete* (✖) displays a `DialogFragment` asking the user to confirm the deletion (Fig. 8.3).

a) Touch the edit icon to edit current contact

b) `Fragment` for editing the contact

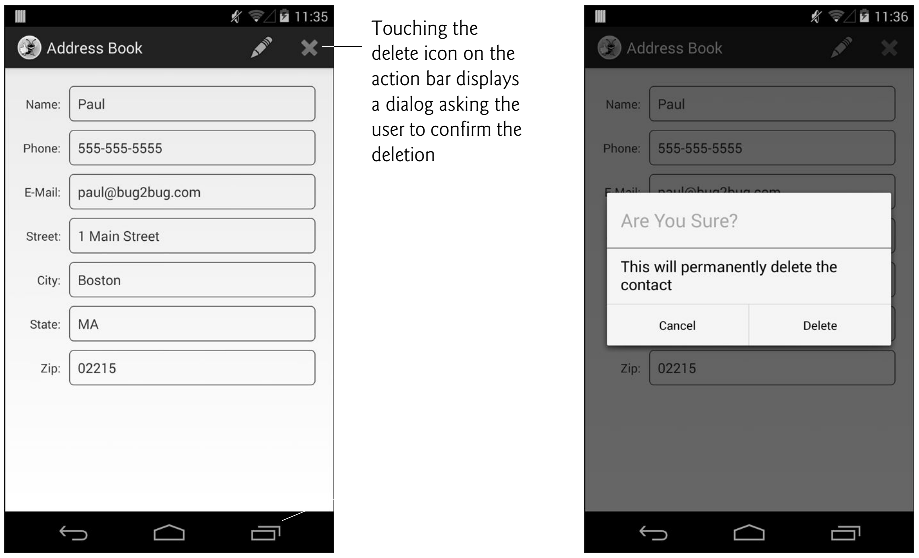

11:35
Address Book
Touching the edit icon on the action bar displays a Fragment for editing that contact's data
Save Contact

Name: Paul
Phone: 555-555-5555
E-Mail: paul@bug2bug.com
Street: 1 Main Street
City: Boston
State: MA
Zip: 02215

Fig. 8.2 | Editing a contact's data.

a) Touch the delete icon to delete current contact

b) Confirmation dialog for deleting contact

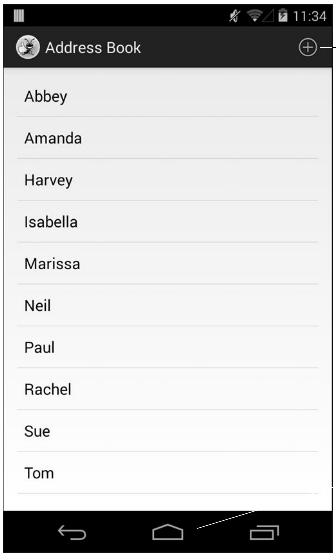

11:35
11:36
Address Book
Touching the delete icon on the action bar displays a dialog asking the user to confirm the deletion
Are You Sure?
Cancel
Delete

This will permanently delete the contact

Fig. 8.3 | Deleting a contact from the database.

When viewing the contact list, touching *add* (⊕) displays a `Fragment` containing `EditTexts` that you can use to add the new contact's data (Fig. 8.4). When editing an existing contact or adding a new one, you touch the **Save Contact** `Button` to save the contact's data. Figure 8.5 shows the app running on a tablet in landscape orientation. On tablets, the contact list is always displayed at the app's left side.

a) Touch the add icon to add a new contact



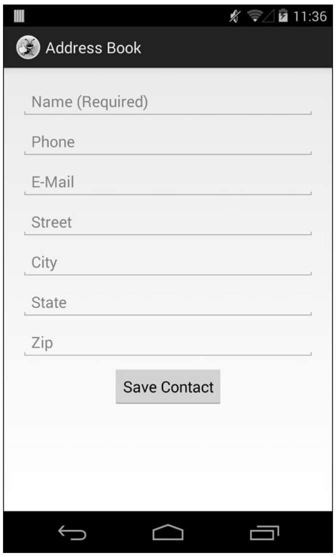
11:34

Address Book

Abbey
Amanda
Harvey
Isabella
Marissa
Neil
Paul
Rachel
Sue
Tom

Touching the add icon on the action bar displays a `Fragment` for entering a new contact

b) `Fragment` for adding the contact



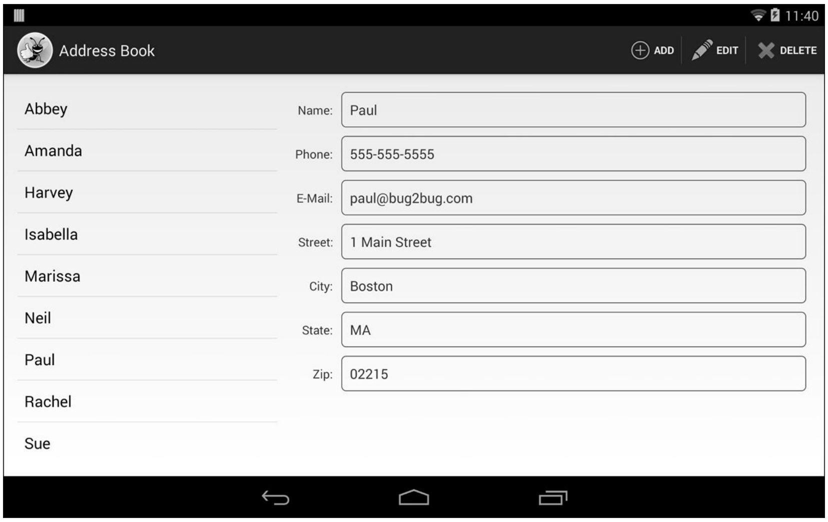
11:36

Address Book

Name (Required)
Phone
E-Mail
Street
City
State
Zip

Fig. 8.4 | Adding a contact to the database.

a) In landscape orientation on a phone or tablet, the action bar icons are displayed with their text



11:40

Address Book

Abbey
Amanda
Harvey
Isabella
Marissa
Neil
Paul
Rachel
Sue

Name: Paul
Phone: 555-555-5555
E-Mail: paul@bug2bug.com
Street: 1 Main Street
City: Boston
State: MA
Zip: 02215

Fig. 8.5 | **Address Book** running in landscape on a tablet.

8.2 Test-Driving the Address Book App

Opening and Running the App

Open Eclipse and import the **Address Book** app project. Perform the following steps:

1. *Open the **Import** Dialog.* Select **File > Import...** to open the **Import** dialog.
2. *Import the **Address Book** app's project.* In the **Import** dialog, expand the **General** node and select **Existing Projects into Workspace**, then click **Next >** to proceed to the **Import Projects** step. Ensure that **Select root directory** is selected, then click the **Browse...** button. In the **Browse for Folder** dialog, locate the `AddressBook` folder in the book's examples folder, select it and click **OK**. Click **Finish** to import the project into Eclipse. The project now appears in the **Package Explorer** window at the left side of the Eclipse window.
3. *Launch the **Address Book** app.* In Eclipse, right click the `AddressBook` project in the **Package Explorer** window, then select **Run As > Android Application** from the menu that appears.

Adding a Contact

The first time you run the app, the contact list will be empty and will display **No Contacts** in the center of the screen. Touch ⊕ on the action bar to display the screen for adding a new entry. After adding the contact's information, touch the **Save Contact** `Button` to store the contact in the database and return to the app's main screen. If you choose not to add the contact, you can simply touch the device's back button to return to the main screen. Add more contacts if you wish. On a tablet, after adding a contact, the new contact's details will be displayed to the right of the contact list, as in Fig. 8.5.

Viewing a Contact

Touch the name of the contact you just added in the contacts list to view that contact's details. On a tablet, the details are displayed to the right of the contact list.

Editing a Contact

While viewing the contact's details, touch ✎ on the action bar to display a screen of `EditTexts` that are prepopulated with the contact's data. Edit the data as necessary, then touch the **Save Contact** `Button` to store the updated contact information in the database and return to the app's main screen. On a tablet, after editing a contact, the new contact's details will be displayed to the right of the contact list.

Deleting a Contact

While viewing the contact's details, touch ✖ on the action bar to delete the contact. A dialog will be displayed asking you to confirm this action. If you do, the contact will be removed from the database and the app will display the updated contact list.

8.3 Technologies Overview

This section presents the new technologies that we use in the **Address Book** app in the order in which they're encountered throughout the chapter.

8.3.1 Displaying `Fragment`s with `FragmentTransaction`s

In earlier apps that used `Fragment`s, you declared each `Fragment` in an `Activity`'s layout or, for a `DialogFragment`, called its show method to create it. The **Flag Quiz** app demonstrated how to use multiple activities to host each of the app's `Fragment`s on a phone device. In this app, you'll use only one `Activity` to host all of the app's `Fragment`s. On a phone-sized device, you'll display one `Fragment` at a time. On a tablet, you'll always display the `Fragment` containing the list of contacts and display the `Fragment`s for viewing, adding and editing contacts as necessary at the app's right side. You'll use the `FragmentManager` and **`FragmentTransaction`s** to dynamically display `Fragment`s. In addition, you'll use Android's `Fragment` **back stack**—a data structure that stores `Fragment`s in last-in-first-out (LIFO) order—to provide automatic support for the Android system bar's back button and to allow the app to remove `Fragment`s in the reverse order from which they were added.

8.3.2 Communicating Data Between a `Fragment` and a Host `Activity`

To communicate data between `Fragment`s and a host `Activity` or the `Activity`'s other `Fragment`s, it's considered best practice to do so through the host `Activity`—this makes the `Fragment`s more reusable, because they do not refer to one another directly. Typically, each `Fragment` defines an *interface* of *callback methods* that are implemented in the host `Activity`. We'll use this technique to enable this app's `MainActivity` to be notified when the user selects a contact to display, touches an action bar item (⊕, ✎ or ✖), or finishes editing an existing contact or adding a new one.

8.3.3 Method `onSaveInstanceState`

`onSaveInstanceState` is called by the system when the configuration of the device changes during the app's execution—for example, when the user rotates the device or slides out a keyboard on a device with a hard keyboard. This method can be used to save state information that you'd like to restore when the app's `onCreate` method is called as part of the configuration change. When an app is simply placed into the background, perhaps so the user can answer a phone call or when the user starts another app, the app's GUI components will automatically save their contents for when the app is brought back to the foreground (provided that the system does not kill the app). We use `onSaveInstanceState` in Fig. 8.47.

8.3.4 Defining Styles and Applying Them to GUI Components

You can define common GUI component attribute–value pairs as **`style` resources** (Section 8.4.4). You can then apply the styles to all components that share those values (Section 8.4.7) by using the **`style` attribute**. Any subsequent changes you make to a `style` are automatically applied to all GUI components that use the `style`. We use this to style the `TextView`s that display a contact's information.

8.3.5 Specifying a Background for a `TextView`

By default `TextView`s do not have a border. To define one, you can specify a `Drawable` as the value for the `TextView`'s `android:background` attribute. The `Drawable` could be an image, but in this app you'll define a `Drawable` as a `shape` in a resource file (Section 8.4.5). The resource file for such a `Drawable` is defined in one or more of the app's `drawable` folders—in this app, `textview_border.xml` is defined in the `drawable-mdpi` folder.

8.3.6 Extending Class `ListFragment` to Create a `Fragment` That Contains a `ListView`

When a `Fragment`'s primary task is to display a scrollable list of items, you can extend class **`ListFragment`** (package `android.app`, Section 8.6)—this is nearly identical to extending `ListActivity`, as you did in Chapter 4. A `ListFragment` uses a `ListView` as its default layout. In this app, rather than an `ArrayAdapter`, we'll use a **`CursorAdapter`** (package `android.widget`) to display the results of a database query in the `ListView`.

8.3.7 Manipulating a SQLite Database

The contact information is stored in a SQLite database. According to `www.sqlite.org`, SQLite is one of the world's most widely deployed database engines. Each `Fragment` in this app interacts with a SQLite database via utility class `DatabaseConnector` (Section 8.9). That class uses a nested subclass of **`SQLiteOpenHelper`** (package **`android.database.sqlite`**), which simplifies creating the database and enables you to obtain a **`SQLiteDatabase`** object (package `android.database.sqlite`) for manipulating a database's contents. Database queries are performed with Structured Query Language (SQL) and query results are managed via a **`Cursor`** (package **`android.database`**).

8.3.8 Performing Database Operations Outside the GUI Thread with `AsyncTask`s

You should perform *long-running operations* or operations that *block* execution until they complete (e.g., file and database access) *outside* the GUI thread. This helps maintain application responsiveness and avoid *Activity Not Responding (ANR) dialogs* that appear when Android thinks the GUI is not responsive. When we need a database operation's results in the GUI thread, we'll use a subclass of **`AsyncTask`** (package `android.os`) to perform the operation in one thread and receive the results in the GUI thread. The details of creating and manipulating threads are handled for you by class `AsyncTask`, as are communicating the results from the `AsyncTask` to the GUI thread.

8.4 Building the GUI and Resource Files

In this section, you'll create the **Address Book** app's additional Java source-code files, resource files and GUI layout files.

8.4.1 Creating the Project

Begin by creating a new Android project. Specify the following values in the **New Android Project** dialog, then press **Finish**:

- **Application Name:** `Address Book`
- **Project Name:** `AddressBook`
- **Package Name:** `com.deitel.addressbook`
- **Minimum Required SDK:** `API18: Android 4.3`
- **Target SDK:** `API19: Android 4.4`
- **Compile With:** `API19: Android 4.4`
- **Theme:** `Holo Light with Dark Action Bar`

In the **New Android Project** dialog's second **New Android Application** step, leave the default settings, and press **Next >**. In the **Configure Launcher Icon** step, select an app icon image, then press **Next >**. In the **Create Activity** step, select **Blank Activity**, then press **Next >**. In the **Blank Activity** step, leave the default settings and click **Finish** to create the project. Open `activity_main.xml` in the **Graphical Layout** editor and select **Nexus 4** from the screen-type drop-down list. Once again, we'll use this device as the basis for our design.

8.4.2 Creating the App's Classes

This app consists of five classes:

- Class `MainActivity` (Section 8.5) manages the app's fragments and coordinates the interactions between them.
- Class `ContactListFragment` (Section 8.6) is a subclass of `ListFragment` that displays the contacts' names and provides a menu item for adding a new contact.
- Class `AddEditFragment` (Section 8.7) is a subclass of `Fragment` that provides a GUI for adding a new contact or editing an existing one.
- Class `DetailsFragment` (Section 8.8) is a subclass of `Fragment` that displays one contact's data and provides menu items for editing and deleting that contact.
- Class `DatabaseConnector` (Section 8.9) is a subclass of **Object** that manages this app's interactions with a SQLite database.

Class `MainActivity` is created by the IDE when you create your project. As you've done in prior projects, you must add the other classes to the project's `com.deitel.addressbook` package in the `src` folder. To do so for each class, right click the package and select **New > Class**, then specify the class's name and superclass.

8.4.3 `strings.xml`

Figure 8.6 shows this app's `String` resource names and corresponding values. Double click `strings.xml` in the `res/values` folder to display the resource editor for creating these `String` resources.

Resource name	Value
`no_contacts`	`No Contacts`
`menuitem_add`	`Add`
`menuitem_edit`	`Edit`
`menuitem_delete`	`Delete`
`button_save_contact`	`Save Contact`
`hint_name`	`Name (Required)`
`hint_email`	`E-Mail`
`hint_phone`	`Phone`
`hint_street`	`Street`
`hint_city`	`City`

Fig. 8.6 | `String` resources used in the **Address Book** app. (Part 1 of 2.)

Resource name	Value
hint_state	State
hint_zip	Zip
label_name	Name:
label_email	E-Mail:
label_phone	Phone:
label_street	Street:
label_city	City:
label_state	State:
label_zip	Zip:
confirm_title	Are You Sure?
confirm_message	This will permanently delete the contact
ok	OK
error_message	You must enter a contact name
button_cancel	Cancel
button_delete	Delete

Fig. 8.6 | `String` resources used in the **Address Book** app. (Part 2 of 2.)

8.4.4 `styles.xml`

In this section, you'll define the styles for the `DetailsFragment`'s `TextViews` that display a contact's information (Section 8.4.7). Like other resources, style resources are placed in the app's `res/values` folder. When you create a project, the IDE creates a `styles.xml` file containing predefined styles. Each new style you create specifies a name that's used to apply that style to GUI components and one or more items specifying property values to apply. To create the new styles:

1. In the app's `res/values` folder, open the `styles.xml` file and ensure that the **Resources** tab is selected at the bottom of the editor window.
2. Click **Add...**, then select **Style/Theme** and click **OK** to create a new style.
3. Set the style's **Name** to `ContactLabelTextView` and save the file.
4. With the `ContactLabelTextView` style selected, click **Add...**, then click **OK** to add an **Item** to the style. Set the **Name** and **Value** attributes for the new **Item** and save the file. Repeat this step for each **Name** and **Value** in Fig. 8.7.

Name	Value
android:layout_width	wrap_content
android:layout_height	wrap_content
android:layout_gravity	right|center_vertical

Fig. 8.7 | `ContactLabelTextView` style attributes.

5. Repeat Steps 2 and 3 to create a style named `ContactTextView`—when you click **Add...**, you'll need to select **Create a new element at the top level in Resources**. Then repeat Step 4 for each **Name** and **Value** in Fig. 8.8. When you're done, save and close `styles.xml`.

Name	Value
`android:layout_width`	`wrap_content`
`android:layout_height`	`wrap_content`
`android:layout_gravity`	`fill_horizontal`
`android:textSize`	`16sp`
`android:background`	`@drawable/textview_border`

Fig. 8.8 | `ContactTextView` style attributes.

8.4.5 `textview_border.xml`

The `style` `ContactTextView` that you created in the preceding section defines the appearance of the `TextView`s that are used to display a contact's details. You specified a `Drawable` (i.e., an image or graphic) named `@drawable/textview_border` as the value for the `TextView`'s `android:background` attribute. In this section, you'll define that `Drawable` in the app's `res/drawable-mdpi` folder. If a `Drawable` is defined in only one of the project's `drawable` folders, Android will use that `Drawable` on *all* device sizes and resolutions. To define the `Drawable`:

1. Right click the `res/drawable-mdpi` folder and select **New > Android XML File**.
2. Specify `textview_border.xml` as the **File** name and select **shape** as the root element, then click **Finish**.
3. At the time of this writing, the IDE does not provide an editor for creating `Drawables`, so enter the XML code in Fig. 8.9 into the file.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <shape xmlns:android="http://schemas.android.com/apk/res/android"
3     android:shape="rectangle" >
4     <corners android:radius="5dp"/>
5     <stroke android:width="1dp" android:color="#555"/>
6     <padding android:top="10dp" android:left="10dp" android:bottom="10dp"
7        android:right="10dp"/>
8  </shape>
```

Fig. 8.9 | XML representation of a `Drawable` that's used to place a border on a `TextView`.

The **shape element**'s element's `android:shape` attribute (line 3) can have the value `"rectangle"` (used in this example), `"oval"`, `"line"` or `"ring"`. The **corners element** (line 4) specifies the rectangle's corner radius, which rounds the corners. The **`stroke` element** (line 5) defines the rectangle's line width and line color. The **`padding` element** (lines 6–7) specifies the spacing around the content in the element to which this `Drawable` is applied. You must specify the top, left, right and bottom padding amounts separately. The complete details of defining shapes can be viewed at:

```
    http://developer.android.com/guide/topics/resources/
       drawable-resource.html#Shape
```

8.4.6 `MainActivity`'s Layout: `activity_main.xml`

You'll provide two layouts for `MainActivity`—one for phone-sized devices in the `res/layout` folder and one for tablet-sized devices in the `res/layout-large` folder. You'll need to add the `layout-large` folder.

Phone Layout: `activity_main.xml` *in* `res/layout`

For the phone layout, open `activity_main.xml` in the `res/layout` folder. Set the `FrameLayout`'s **Id** to `@id/fragmentContainer`. This `FrameLayout` will be used on phones to display the app's `Fragments`. Set the **Padding Left**, **Padding Right**, **Padding Top** and **Padding Bottom** properties for the `FrameLayout` as you did for other layouts in earlier chapters.

Tablet Layout: `activity_main.xml` *in* `res/layout-large`

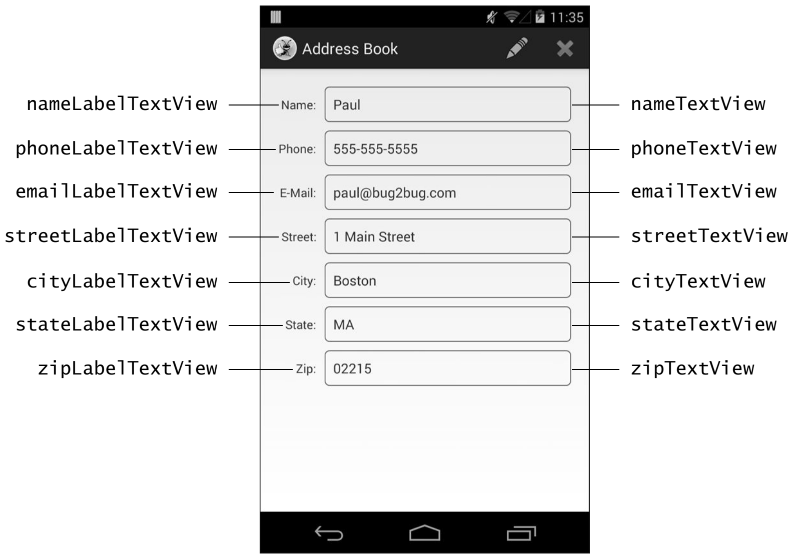
For the tablet layout, create a new `activity_main.xml` layout in the `res/layout-large` folder. This layout should use a horizontal `LinearLayout` containing a `ContactListFragment` and an empty `FrameLayout`. Use the techniques you learned in Section 5.4.9 to add the `ContactListFragment` to the layout, then add the `FrameLayout`. Set the following properties:

- For the `LinearLayout` set **Weight Sum** to `3`—this will help allocate the horizontal space to the `ContactListFragment` and `FrameLayout`.
- For the `Fragment`, set the **Id** to `@+id/contactListFragment`, the **Width** to `0`, the **Height** to `match_parent`, the **Weight** to `1` and the **Right** margin to `@dimen/activity_horizontal_margin`.
- For the `FrameLayout` set the **Id** to `@+id/rightPaneContainer`, the **Width** to `0`, the **Height** to `match_parent` and the **Weight** to `2`.

Setting the `LinearLayout`'s **Weight Sum** to `3`, then setting the `ContactListFragment`'s and `FrameLayout`'s **Weight**s to `1` and `2`, respectively, indicates that the `ContactListFragment` should occupy one-third of the `LinearLayout`'s width and the `FrameLayout` should occupy the remaining two-thirds.

8.4.7 `DetailsFragment`'s Layout: `fragment_details.xml`

When the user touches a contact in the `MainActivity`, the app displays the `DetailsFragment` (Fig. 8.10). This `Fragment`'s layout (`fragment_details.xml`) consists of a `ScrollView` containing a vertical `GridLayout` with two columns of `TextViews`. A **`ScrollView`** is a **`ViewGroup`** that can contain other `Views` (like a layout) and that lets users *scroll* through content too large to display on the screen. We use a `ScrollView` here to ensure that the user can scroll through a contact's details if a device does not have enough vertical space to show all the `TextViews` in Fig. 8.10. Follow the steps in Section 5.4.8 to create the `fragment_details.xml` file, but use a `ScrollView` as the **Root Element**. After creating the file, set the `ScrollView`'s **Id** to `@+id/detailsScrollView` and add a `GridLayout` to the `ScrollView`.


Address Book

Name: Paul

Phone: 555-555-5555

E-Mail: paul@bug2bug.com

Street: 1 Main Street

City: Boston

State: MA

Zip: 02215

Fig. 8.10 | `DetailsFragment`'s GUI components labeled with their `id` property values.

`GridLayout` Settings
For the `GridLayout`, we set the **Width** to `match_parent`, **Height** to `wrap_content`, **Column Count** to `2` and **Use Default Margins** to `true`. The **Height** value enables the parent `ScrollView` to determine the `GridLayout`'s actual height and decide whether to provide scrolling. Add `TextView`s to the `GridLayout` as shown in Fig. 8.10.

Left Column `TextView` Settings
For each `TextView` in the left column set the `TextView`'s **Id** property as specified in Fig. 8.10 and set:

- **Row** to a value from 0–6 depending on the row.
- **Column** to `0`.
- **Text** to the appropriate `String` resource from `strings.xml`.
- **Style** (located in the **View** category) to `@style/ContactLabelTextView`—style resources are specified using the syntax `@style/`*styleName*.

Right Column `TextView` Settings
For each `TextView` in the right column set the `TextView`'s **Id** property as specified in Fig. 8.10 and set:

- **Row** to a value from 0–6 depending on the row.
- **Column** to `1`.
- **Style** (located in the **View** category) to `@style/ContactTextView`.

8.4.8 `AddEditFragment`'s Layout: `fragment_add_edit.xml`

When the user touches the action bar items ⊕ or ✎, the `MainActivity` displays the `AddEditFragment` (Fig. 8.11) with a layout (`fragment_add_edit.xml`) that uses a `ScrollView` containing a one-column vertical `GridLayout`. Be sure to set the `ScrollView`'s **Id** to `@+id/addEditScrollView`. If the `AddEditFragment` is displayed to add a new contact, the `EditTexts` will be empty and will display *hints* (Fig. 8.4). Otherwise, they'll display the contact's data that was passed to the `AddEditFragment` by the `MainActivity`. Each `EditText` specifies the **Input Type** and **IME Options** properties. For devices that display a soft keyboard, the **Input Type** specifies which keyboard to display when the user touches the corresponding `EditText`. This enables us to *customize the keyboard* to the specific type of data the user must enter in a given `EditText`. We use the **IME Options** property to display a **Next** button on the soft keyboards for the `nameEditText`, `emailEditText`, `phoneEditText`, `streetEditText`, `cityEditText` and `stateEditText`. When one of these has the focus, touching this `Button` transfers the focus to the next `EditText`. If the `zipEditText` has the focus, you can hide the soft keyboard by touching the keyboard's **Done** `Button`.

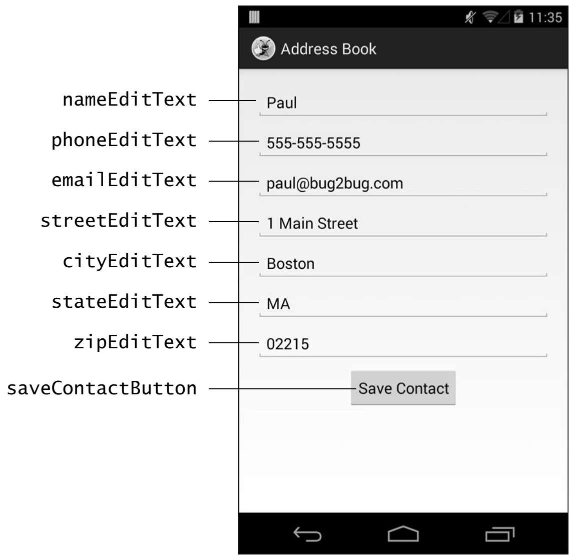


Fig. 8.11 | `AddEditFragment`'s GUI components labeled with their `id` property values. This GUI's root component is a `ScrollView` that contains a vertical `GridLayout`.

`GridLayout` Settings

For the `GridLayout`, we set the **Width** to `match_parent`, **Height** to `wrap_content`, **Column Count** to `1` and **Use Default Margins** to `true`. Add the components shown in Fig. 8.11.

`EditText` Settings

For each `EditText`, set the `TextView`'s **Id** property as specified in Fig. 8.11 and set:

- **Width** to `match_parent`.
- **Height** to `wrap_content`.

- **Hint** to the appropriate `String` resource from `strings.xml`.
- **IME Options** to `actionNext` for all `EditTexts` except `zipEditText`, which should have the value `actionDone`.
- **Style** (located in the **View** category) to `@style/ContactLabelTextView`—style resources are specified using the syntax `@style/`*styleName*.

Set the `EditTexts`' **Input Type** properties to display appropriate keyboards as follows:

- `nameEditText`: `textPersonName|textCapWords`—for entering names and starts each word with a capital letter.
- `phoneEditText`: `phone`—for entering phone numbers.
- `emailEditText`: `textEmailAddress`—for entering an e-mail address.
- `streetEditText`: `textPostalAddress|textCapWords`—for entering an address and starts each word with a capital letter.
- `cityEditText`: `textPostalAddress|textCapWords`.
- `stateEditText`: `textPostalAddress|textCapCharacters`—ensures that state abbreviations are displayed in capital letters.
- `zipEditText`: `number`—for entering numbers.

8.4.9 Defining the `Fragments`' Menus

You'll now use the techniques you learned in Section 7.3.4 to create two menu resource files in the app's res/menu folder:

- `fragment_contact_list_menu.xml` defines the menu item for adding a contact.
- `fragment_details_menu.xml` defines the menu items for editing an existing contact and deleting a contact.

When both the `ContactListFragment` and the `DetailsFragment` are displayed on a tablet at the same time, all of the menu items are displayed.

Figures 8.12–8.13 show the settings for the menu items in the two menu resource files. Each menu item's **Order in category** values determines the order in which the menu items appear on the action bar. For each menu item's **Icon** value, we specified a standard Android icon. You can see the complete set of standard icons in the Android SDK's `platforms` folder under each platform version's `data/res/drawable-hdpi` folder. To refer to these icons in your menus or layouts, prefix them with `@android:drawable/`*icon_name*.

Name	Value
Id	`@id/action_add`
Order in category	0
Title	`@string/menuitem_add`
Icon	`@android:drawable/ic_menu_add`
Show as action	`ifRoom|withText`

Fig. 8.12 | Menu item for `fragment_contact_list_menu.xml`.

Name	Value
Edit menu item	
Id	`@id/action_edit`
Order in category	`1`
Title	`@string/menuitem_edit`
Icon	`@android:drawable/ic_menu_edit`
Show as action	`ifRoom|withText`
Delete menu item	
Id	`@id/action_delete`
Order in category	`2`
Title	`@string/menuitem_delete`
Icon	`@android:drawable/ic_delete`
Show as action	`ifRoom|withText`

Fig. 8.13 | Menu item for `fragment_details_menu.xml`.

8.5 `MainActivity` Class

Class `MainActivity` (Figs. 8.14–8.23) manages the app's fragments and coordinates the interactions between them. On phones, `MainActivity` displays one `Fragment` at a time, starting with the `ContactListFragment`. On tablets, `MainActivity` always displays the `ContactListFragment` at the left of the layout and, depending on the context, displays either the `DetailsFragment` or the `AddEditFragment` in the right two-thirds of the layout.

`MainActivity` `package` Statement, `import` statements and Fields

Class `MainActivity` (Fig. 8.14) uses class `FragmentTransaction` (imported at line 6) to add and remove the app's `Fragments`. `MainActivity` implements three interfaces:

- `ContactListFragment.ContactListFragmentListener` contains callback methods that the `ContactListFragment` uses to tell the `MainActivity` when the user selects a contact in the contact list or adds a new contact.
- `DetailsFragment.DetailsFragmentListener` contains callback methods that the `DetailsFragment` uses to tell the `MainActivity` when the user deletes a contact or wishes to edit an existing contact.
- `AddEditFragment.AddEditFragmentListener` contains callback methods that the `AddEditFragment` uses to tell the `MainActivity` when the user finishes adding a new contact or editing an existing one.

The constant `ROW_ID` (line 15) is used as a key in a key–value pair that's passed between the `MainActivity` and its `Fragments`. The instance variable `contactListFragment` (line 17) is used to tell the `ContactListFragment` to update the displayed list of contacts after a contact is added or deleted.

```
 1  // MainActivity.java
 2  // Hosts Address Book app's fragments
 3  package com.deitel.addressbook;
 4
 5  import android.app.Activity;
 6  import android.app.FragmentTransaction;
 7  import android.os.Bundle;
 8
 9  public class MainActivity extends Activity
10     implements ContactListFragment.ContactListFragmentListener,
11        DetailsFragment.DetailsFragmentListener,
12        AddEditFragment.AddEditFragmentListener
13  {
14     // keys for storing row ID in Bundle passed to a fragment
15     public static final String ROW_ID = "row_id";
16
17     ContactListFragment contactListFragment; // displays contact list
18
```

Fig. 8.14 | `MainActivity` `package` statement, `import` statements and fields.

`MainActivity` Overridden `onCreate` Method

Method `onCreate` (Fig. 8.15) inflates `MainActivity`'s GUI and, if the app is running on a phone-sized device, displays a `ContactListFragment`. As you'll see in Section 8.6, you can configure a `Fragment` to be retained across configuration changes, such as when the user rotates the device. If the `Activity` is being restored after being shut down or recreated from a configuration change, `savedInstanceState` will not be `null`. In this case, we simply return (line 28) because the `ContactListFragment` already exists—on a phone, it would have been retained and on a tablet, it's part of the `MainActivity`'s layout that was inflated in line 24.

```
19     // display ContactListFragment when MainActivity first loads
20     @Override
21     protected void onCreate(Bundle savedInstanceState)
22     {
23        super.onCreate(savedInstanceState);
24        setContentView(R.layout.activity_main);
25
26        // return if Activity is being restored, no need to recreate GUI
27        if (savedInstanceState != null)
28           return;
29
30        // check whether layout contains fragmentContainer (phone layout);
31        // ContactListFragment is always displayed
32        if (findViewById(R.id.fragmentContainer) != null)
33        {
34           // create ContactListFragment
35           contactListFragment = new ContactListFragment();
36
```

Fig. 8.15 | `MainActivity` overridden `onCreate` method. (Part 1 of 2.)

```
37            // add the fragment to the FrameLayout
38            FragmentTransaction transaction =
39               getFragmentManager().beginTransaction();
40            transaction.add(R.id.fragmentContainer, contactListFragment);
41            transaction.commit(); // causes ContactListFragment to display
42         }
43      }
44
```

Fig. 8.15 | `MainActivity` overridden `onCreate` method. (Part 2 of 2.)

If the `R.id.fragmentContainer` exists in `MainActivity`'s layout (line 32), then the app is running on a phone. In this case, line 35 creates the `ContactListFragment`, then lines 38–41 use a `FragmentTransaction` to add the `ContactListFragment` to the user interface. Lines 38–39 call `FragmentManager`'s **`beginTransaction`** method to obtain a `FragmentTransaction`. Next, line 40 uses `FragmentTransaction` method **`add`** to specify that, when the `FragmentTransaction` completes, the `ContactListFragment` should be attached to the `View` with the ID specified as the first argument. Finally, line 41 uses `FragmentTransaction` method **`commit`** to finalize the transaction and display the `ContactListFragment`.

`MainActivity` Overridden `onResume` Method

Method `onResume` (Fig. 8.16) determines whether `contactListFragment` is `null`—if so, the app is running on a tablet, so lines 55–57 use the `FragmentManager` to get a reference to the existing `ContactListFragment` in `MainActivity`'s layout.

```
45      // called when MainActivity resumes
46      @Override
47      protected void onResume()
48      {
49         super.onResume();
50
51         // if contactListFragment is null, activity running on tablet,
52         // so get reference from FragmentManager
53         if (contactListFragment == null)
54         {
55            contactListFragment =
56               (ContactListFragment) getFragmentManager().findFragmentById(
57                  R.id.contactListFragment);
58         }
59      }
60
```

Fig. 8.16 | `MainActivity` overridden `onResume` method.

`MainActivity` Method `onContactSelected`

Method `onContactSelected` (Fig. 8.17) from the `ContactListFragment.ContactListFragmentListener` interface is called by the `ContactListFragment` to notify the `MainActivity` when the user selects a contact to display. If the app is running on a phone (line 65), line 66 calls method `displayContact` (Fig. 8.18), which replaces the `ContactListFragment` in the fragmentContainer (defined in Section 8.4.6) with the DetailsFragment that shows the contact's information. On a tablet, line 69 calls the FragmentManager's **popBackStack** method to *pop* (remove) the top Fragment on the back stack, then line 70 calls displayContact, which replaces the contents of the rightPaneContainer (defined in Section 8.4.6) with the DetailsFragment that shows the contact's information.

```
61      // display DetailsFragment for selected contact
62      @Override
63      public void onContactSelected(long rowID)
64      {
65         if (findViewById(R.id.fragmentContainer) != null) // phone
66            displayContact(rowID, R.id.fragmentContainer);
67         else // tablet
68         {
69            getFragmentManager().popBackStack(); // removes top of back stack
70            displayContact(rowID, R.id.rightPaneContainer);
71         }
72      }
73
```

Fig. 8.17 | MainActivity method onContactSelected.

MainActivity Method displayContact

Method displayContact (Fig. 8.18) creates the DetailsFragment that displays the selected contact and uses a FragmentTransaction to attach it to the GUI. You can pass arguments to a Fragment by placing them in a Bundle of key–value pairs—we do this to pass the selected contact's rowID so that the DetailsFragment knows which contact to get from the database. Line 80 creates the Bundle. Line 81 calls its **putLong** method to store a key–value pair containing the ROW_ID (a String) as the key and the rowID (a long) as the value. Line 82 passes the Bundle to the Fragment's **setArguments** method—the Fragment can then extract the information from the Bundle (as you'll see in Section 8.8). Lines 85–86 get a FragmentTransaction, then line 87 calls FragmentTransaction method **replace** to specify that, when the FragmentTransaction completes, the DetailsFragment should replace the contents of the View with the ID specified as the first argument. Line 88 calls FragmentTransaction method **addToBackStack** to *push* (add) the DetailsFragment onto the back stack. This allows the user to touch the back button to pop the Fragment from the back stack and allows MainActivity to programmatically pop the Fragment from the back stack.

```
74      // display a contact
75      private void displayContact(long rowID, int viewID)
76      {
77         DetailsFragment detailsFragment = new DetailsFragment();
78
79         // specify rowID as an argument to the DetailsFragment
80         Bundle arguments = new Bundle();
81         arguments.putLong(ROW_ID, rowID);
82         detailsFragment.setArguments(arguments);
```

Fig. 8.18 | MainActivity method displayContact. (Part 1 of 2.)

```
83
84          // use a FragmentTransaction to display the DetailsFragment
85          FragmentTransaction transaction =
86             getFragmentManager().beginTransaction();
87          transaction.replace(viewID, detailsFragment);
88          transaction.addToBackStack(null);
89          transaction.commit(); // causes DetailsFragment to display
90       }
91
```

Fig. 8.18 | `MainActivity` method `displayContact`. (Part 2 of 2.)

`MainActivity` Method `onAddContact`

Method `onAddContact` (Fig. 8.19) from the `ContactListFragment.ContactListFragmentListener` interface is called by the `ContactListFragment` to notify the `MainActivity` when the user chooses to add a new contact. If the layout contains the `fragmentContainer`, line 97 calls `displayAddEditFragment` (Fig. 8.20) to display the `AddEditFragment` in the `fragmentContainer`; otherwise, line 99 calls `displayAddEditFragment` to display the `Fragment` in the `rightPaneContainer`. The second argument is a `Bundle`. Specifying `null` indicates that a new contact is being added.

```
92       // display the AddEditFragment to add a new contact
93       @Override
94       public void onAddContact()
95       {
96          if (findViewById(R.id.fragmentContainer) != null) // phone
97             displayAddEditFragment(R.id.fragmentContainer, null);
98          else // tablet
99             displayAddEditFragment(R.id.rightPaneContainer, null);
100      }
101
```

Fig. 8.19 | `MainActivity` method `onAddContact`.

`MainActivity` Method `displayAddEditFragment`

Method `displayAddEditFragment` (Fig. 8.20) receives a `View`'s resource ID specifying where to attach the `AddEditFragment` and a `Bundle` of key–value pairs. If the second argument is `null`, a new contact is being added; otherwise, the `Bundle` contains the data to display in the `AddEditFragment` for editing. Line 105 creates the `AddEditFragment`. If the `Bundle` argument is not `null`, line 108 uses it to set the `Fragment`'s arguments. Lines 111–115 then create the `FragmentTransaction`, replace the contents of the `View` with the specified resource ID, add the `Fragment` to the back stack and commit the transaction.

```
102      // display fragment for adding a new or editing an existing contact
103      private void displayAddEditFragment(int viewID, Bundle arguments)
104      {
105         AddEditFragment addEditFragment = new AddEditFragment();
```

Fig. 8.20 | `MainActivity` Method `displayAddEditFragment`. (Part 1 of 2.)

```
106
107          if (arguments != null) // editing existing contact
108             addEditFragment.setArguments(arguments);
109
110          // use a FragmentTransaction to display the AddEditFragment
111          FragmentTransaction transaction =
112             getFragmentManager().beginTransaction();
113          transaction.replace(viewID, addEditFragment);
114          transaction.addToBackStack(null);
115          transaction.commit(); // causes AddEditFragment to display
116       }
117
```

Fig. 8.20 | MainActivity Method displayAddEditFragment. (Part 2 of 2.)

MainActivity Method onContactDeleted

Method onContactDeleted (Fig. 8.21) from the DetailsFragment.DetailsFragmentListener interface is called by the DetailsFragment to notify the MainActivity when the user deletes a contact. In this case, line 122 pops the DetailsFragment from the back stack. If the app is running on a tablet, line 125 calls the contactListFragment's updateContactList method to reload the contacts.

```
118       // return to contact list when displayed contact deleted
119       @Override
120       public void onContactDeleted()
121       {
122          getFragmentManager().popBackStack(); // removes top of back stack
123
124          if (findViewById(R.id.fragmentContainer) == null) // tablet
125             contactListFragment.updateContactList();
126       }
127
```

Fig. 8.21 | MainActivity method onContactDeleted.

MainActivity Method onEditContact

Method onEditContact (Fig. 8.22) from the DetailsFragment.DetailsFragmentListener interface is called by the DetailsFragment to notify the MainActivity when the user touches the menu item to edit a contact. The DetailsFragment passes a Bundle containing the contact's data so that it can be displayed in the AddEditFragment's EditTexts for editing. If the layout contains the fragmentContainer, line 133 calls displayAddEditFragment to display the AddEditFragment in the fragmentContainer; otherwise, line 135 calls displayAddEditFragment to display the AddEditFragment in the rightPaneContainer.

```
128       // display the AddEditFragment to edit an existing contact
129       @Override
130       public void onEditContact(Bundle arguments)
131       {
```

Fig. 8.22 | MainActivity method onEditContact. (Part 1 of 2.)

```
      if (findViewById(R.id.fragmentContainer) != null) // phone
         displayAddEditFragment(R.id.fragmentContainer, arguments);
      else // tablet
         displayAddEditFragment(R.id.rightPaneContainer, arguments);
   }

```

Fig. 8.22 | `MainActivity` method `onEditContact`. (Part 2 of 2.)

`MainActivity` *Method* ***`onAddEditCompleted`***

Method `onAddEditCompleted` (Fig. 8.23) from the `AddEditFragment.AddEditFragment-Listener` interface is called by the `AddEditFragment` to notify the `MainActivity` when the user saves a new contact or saves changes to an existing one. Line 142 pops the `AddEditFragment` from the back stack. If the app is running on a tablet (line 144), line 146 pops the back stack again to remove the `DetailsFragment` (if there is one). Then line 147 updates the contact list in the `ContactListFragment` and line 150 displays the new or updated contact's details in the `rightPaneContainer`.

```
138    // update GUI after new contact or updated contact saved
139    @Override
140    public void onAddEditCompleted(long rowID)
141    {
142       getFragmentManager().popBackStack(); // removes top of back stack
143
144       if (findViewById(R.id.fragmentContainer) == null) // tablet
145       {
146          getFragmentManager().popBackStack(); // removes top of back stack
147          contactListFragment.updateContactList(); // refresh contacts
148
149          // on tablet, display contact that was just added or edited
150          displayContact(rowID, R.id.rightPaneContainer);
151       }
152    }
153 }
```

Fig. 8.23 | `MainActivity` method `onAddEditCompleted`.

8.6 `ContactListFragment` Class

Class `ContactListFragment` (Figs. 8.24–8.33) extends `ListFragment` to display the contact list in a `ListView` and provides a menu item for adding a new contact.

`ContactListFragment` ***`package`*** *Statement and* ***`import`*** *Statements*

Figure 8.24 lists `ContactListFragment`'s `package` statement and `import` statements. We've highlighted the `imports` for the new classes and interfaces.

```
 1  // ContactListFragment.java
 2  // Displays the list of contact names
 3  package com.deitel.addressbook;
 4
 5  import android.app.Activity;
 6  import android.app.ListFragment;
 7  import android.database.Cursor;
 8  import android.os.AsyncTask;
 9  import android.os.Bundle;
10  import android.view.Menu;
11  import android.view.MenuInflater;
12  import android.view.MenuItem;
13  import android.view.View;
14  import android.widget.AdapterView;
15  import android.widget.AdapterView.OnItemClickListener;
16  import android.widget.CursorAdapter;
17  import android.widget.ListView;
18  import android.widget.SimpleCursorAdapter;
19
```

Fig. 8.24 | `ContactListFragment` `package` statement and `import` statements.

`ContactListFragmentListener` Interface and `ContactListFragment` Instance Variables

Figure 8.25 begins class `ContactListFragment`'s declaration. Lines 23–30 declare the nested interface `ContactListFragmentListener`, which contains the callback methods that `MainActivity` implements to be notified when the user selects a contact (line 26) and when the user touches the menu item to add a new contact (line 29). Line 32 declares instance variable `listener` which will refer to the object (`MainActivity`) that implements the interface. Instance variable `contactListView` (line 34) will refer to the `ContactListFragment`'s built-in `ListView`, so we can interact with it programmatically. Instance variable `contactAdapter` will refer to the `CursorAdapter` that populates the `AddressBook`'s `ListView`.

```
20  public class ContactListFragment extends ListFragment
21  {
22     // callback methods implemented by MainActivity
23     public interface ContactListFragmentListener
24     {
25        // called when user selects a contact
26        public void onContactSelected(long rowID);
27
28        // called when user decides to add a contact
29        public void onAddContact();
30     }
31
32     private ContactListFragmentListener listener;
33
```

Fig. 8.25 | `ContactListFragmentListener` interface and `ContactListFragment` instance variables. (Part 1 of 2.)

```
34      private ListView contactListView; // the ListActivity's ListView
35      private CursorAdapter contactAdapter; // adapter for ListView
36
```

Fig. 8.25 | `ContactListFragmentListener` interface and `ContactListFragment` instance variables. (Part 2 of 2.)

`ContactListFragment` Overridden Methods `onAttach` and `onDetach`

Class `ContactListFragment` overrides `Fragment` lifecycle methods `onAttach` and `onDetach` (Fig. 8.26) to set instance variable `listener`. In this app, `listener` refers to the host `Activity` (line 42) when the `ContactListFragment` is attached and is set to `null` (line 50) when the `ContactListFragment` is detached.

```
37      // set ContactListFragmentListener when fragment attached
38      @Override
39      public void onAttach(Activity activity)
40      {
41         super.onAttach(activity);
42         listener = (ContactListFragmentListener) activity;
43      }
44
45      // remove ContactListFragmentListener when Fragment detached
46      @Override
47      public void onDetach()
48      {
49         super.onDetach();
50         listener = null;
51      }
52
```

Fig. 8.26 | `ContactListFragment` overridden methods `onAttach` and `onDetach`.

`ContactListFragment` Overridden Method `onViewCreated`

Recall that class `ListFragment` already contains a `ListView`, so we don't need to inflate the GUI as in previous app's `Fragments`. However, class `ContactListFragment` has tasks that should be performed after its default layout is inflated. For this reason, `ContactListFragment` overrides `Fragment` lifecycle method **`onViewCreated`** (Fig. 8.27), which is called after `onCreateView`.

```
53      // called after View is created
54      @Override
55      public void onViewCreated(View view, Bundle savedInstanceState)
56      {
57         super.onViewCreated(view, savedInstanceState);
58         setRetainInstance(true); // save fragment across config changes
59         setHasOptionsMenu(true); // this fragment has menu items to display
60
```

Fig. 8.27 | `ContactListFragment` overridden method `onViewCreated`. (Part 1 of 2.)

```
61          // set text to display when there are no contacts
62          setEmptyText(getResources().getString(R.string.no_contacts));
63
64          // get ListView reference and configure ListView
65          contactListView = getListView();
66          contactListView.setOnItemClickListener(viewContactListener);
67          contactListView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
68
69          // map each contact's name to a TextView in the ListView layout
70          String[] from = new String[] { "name" };
71          int[] to = new int[] { android.R.id.text1 };
72          contactAdapter = new SimpleCursorAdapter(getActivity(),
73             android.R.layout.simple_list_item_1, null, from, to, 0);
74          setListAdapter(contactAdapter); // set adapter that supplies data
75       }
76
```

Fig. 8.27 | `ContactListFragment` overridden method `onViewCreated`. (Part 2 of 2.)

Line 58 calls `Fragment` method **`setRetainInstance`** with the argument `true` to indicate that the `ContactListFragment` should be retained rather than recreated when the host `Activity` is re-created on a configuration change (e.g., when the user rotates the device). Line 59 indicates that the `ContactListFragment` has menu items that should be displayed on the `Activity`'s action bar (or in its options menu). `ListFragment` method **`setEmptyText`** (line 62) specifies the text to display (`"No Contacts"`) when there are no items in the `ListView`'s adapter.

Line 65 uses the inherited `ListActivity` method **`getListView`** to obtain a reference to the built-in `ListView`. Line 66 sets the `ListView`'s `OnItemClickListener` to `viewContactListener` (Fig. 8.28), which responds when the user touches a contact in the `ListView`. Line 67 calls `ListView` method **`setChoiceMode`** to indicate that only one item can be selected at a time.

*Configuring the **`CursorAdapter`** That Binds Database Data to the **`ListView`***

To display the `Cursor`'s results in a `ListView` we create a new `CursorAdapter` object (lines 70–73) which exposes the `Cursor`'s data in a manner that can be used by a `ListView`. **`SimpleCursorAdapter`** is a subclass of `CursorAdapter` that's designed to simplify mapping `Cursor` columns directly to `TextView`s or `ImagesView`s defined in your XML layouts. To create a `SimpleCursorAdapter`, you first define arrays containing the column names to map to GUI components and the resource IDs of the GUI components that will display the data from the named columns. Line 70 creates a `String` array indicating that only the `"name"` column will be displayed, and line 71 creates a parallel `int` array containing corresponding GUI components' resource IDs. Chapter 4 showed that you can create your own layout resources for `ListView` items. In this app we used a predefined Android layout resource named `android.R.layout.simple_list_item_1`—a layout that contains one `TextView` with the ID `android.R.id.text1`. Lines 72–73 create the `SimpleCursorAdapter`. Its constructor receives:

- the `Context` in which the `ListView` is running (i.e., `MainActivity`).
- the resource ID of the layout that's used to display each item in the `ListView`.

- the `Cursor` that provides access to the data—we supply `null` for this argument because we'll specify the `Cursor` later.
- the `String` array containing the column names to display.
- the `int` array containing the corresponding GUI resource IDs.
- the last argument is typically 0.

Line 74 uses inherited `ListActivity` method **`setListAdapter`** to bind the `ListView` to the `CursorAdapter`, so that the `ListView` can display the data.

`viewContactListener` That Processes `ListView` Item Selection Events

The `viewContactListener` (Fig. 8.28) notifies `MainActivity` when the user touches a contact to display. Line 84 passes the argument `id`—the row ID of the selected contact—to the `listener`'s `onContactSelected` method (Fig. 8.17).

```
77     // responds to the user touching a contact's name in the ListView
78     OnItemClickListener viewContactListener = new OnItemClickListener()
79     {
80        @Override
81        public void onItemClick(AdapterView<?> parent, View view,
82           int position, long id)
83        {
84           listener.onContactSelected(id); // pass selection to MainActivity
85        }
86     }; // end viewContactListener
87
```

Fig. 8.28 | `viewContactListener` that processes `ListView` item selection events.

`ContactListFragment` Overridden Method `onResume`

`Fragment` lifecycle method `onResume` (Fig. 8.29) creates and executes an `AsyncTask` (line 93) of type `GetContactsTask` (defined in Fig. 8.30) that gets the complete list of contacts from the database and sets the `contactAdapter`'s `Cursor` for populating the `ContactListFragment`'s `ListView`. `AsyncTask` method **`execute`** performs the task in a separate thread. Method `execute`'s argument in this case indicates that the task does not receive any arguments—this method can receive a variable number of arguments that are, in turn, passed as arguments to the task's `doInBackground` method. Every time line 93 executes, it creates a new `GetContactsTask` object—this is required because each `AsyncTask` can be executed *only once*.

```
88     // when fragment resumes, use a GetContactsTask to load contacts
89     @Override
90     public void onResume()
91     {
92        super.onResume();
93        new GetContactsTask().execute((Object[]) null);
94     }
95
```

Fig. 8.29 | `ContactListFragment` overridden method `onResume`.

GetContactsTask Subclass of AsyncTask

Nested class GetContactsTask (Fig. 8.30) extends class AsyncTask. The class defines how to interact with the DatabaseConnector (Section 8.9) to get the names of all the contacts and return the results to this Activity's GUI thread for display in the ListView. AsyncTask is a generic type that requires three type parameters:

- The variable-length parameter-list type for AsyncTask's **doInBackground** method (lines 103–108)—When you call the task's execute method, doInBackground performs the task in a separate thread. We specify Object as the type parameter and pass null as the argument to the AsyncTask's execute method, because GetContactsTask does not require additional data to perform its task.
- The variable-length parameter-list type for the AsyncTask's **onProgressUpdate** method—This method executes in the GUI thread and is used to receive *intermediate updates* of the specified type from a long-running task. We don't use this feature in this example, so we specify type Object here and ignore this type parameter.
- The type of the task's result, which is passed to the AsyncTask's **onPostExecute** method (lines 111–116)—This method executes in the GUI thread and enables the ContactListFragment to use the AsyncTask's results.

A key benefit of using an AsyncTask is that it handles the details of creating threads and executing its methods on the appropriate threads for you, so that you do not have to interact with the threading mechanism directly.

```
96     // performs database query outside GUI thread
97     private class GetContactsTask extends AsyncTask<Object, Object, Cursor>
98     {
99        DatabaseConnector databaseConnector =
100          new DatabaseConnector(getActivity());
101
102       // open database and return Cursor for all contacts
103       @Override
104       protected Cursor doInBackground(Object... params)
105       {
106          databaseConnector.open();
107          return databaseConnector.getAllContacts();
108       }
109
110       // use the Cursor returned from the doInBackground method
111       @Override
112       protected void onPostExecute(Cursor result)
113       {
114          contactAdapter.changeCursor(result); // set the adapter's Cursor
115          databaseConnector.close();
116       }
117    } // end class GetContactsTask
118
```

Fig. 8.30 | GetContactsTask subclass of AsyncTask.

Lines 99–100 create a new object of our utility class `DatabaseConnector`, passing the `Context` (the `ContactListFragment`'s host `Activity`) as an argument to the class's constructor. Method `doInBackground` uses `databaseConnector` to open the database connection and get all the contacts from the database. The `Cursor` returned by `getAllContacts` is passed to method `onPostExecute`, which receives the `Cursor` containing the results and passes it to the `contactAdapter`'s `changeCursor` method. This enables the `ContactListFragment`'s `ListView` to populate itself with the contacts' names.

`ContactListFragment` Overridden Method `onStop`

`Fragment` lifecycle method **`onStop`** (Fig. 8.31) is called after `onPause` when the `Fragment` is no longer visible to the user. In this case, the `Cursor` that allows us to populate the `ListView` is not needed, so line 123 calls `CursorAdapter` method **`getCursor`** to get the current `Cursor` from the `contactAdapter`. Line 124 calls `CursorAdapter` method **`changeCursor`** with the argument `null` to remove the `Cursor` from the `CursorAdapter`. Then line 127 calls `Cursor` method **`close`** to release resources used by the `Cursor`.

```
119    // when fragment stops, close Cursor and remove from contactAdapter
120    @Override
121    public void onStop()
122    {
123       Cursor cursor = contactAdapter.getCursor(); // get current Cursor
124       contactAdapter.changeCursor(null); // adapter now has no Cursor
125
126       if (cursor != null)
127          cursor.close(); // release the Cursor's resources
128
129       super.onStop();
130    }
131
```

Fig. 8.31 | `ContactListFragment` overridden method `onStop`.

`ContactListFragment` Overridden Methods `onCreateOptionsMenu` and `onOptionsItemSelected`

Method `onCreateOptionsMenu` (Fig. 8.32, lines 133–138) uses its `MenuInflater` argument to create the menu from `fragment_contact_list_menu.xml`, which contains the definition of the add (⊕) menu item. If the user touches that `MenuItem`, method `onOptionsItemSelected` (lines 141–152) calls `listener`'s `onAddContact` method to notify the `MainActivity` that the user wants to add a new contact. `MainActivity` then displays the `AddEditFragment` (Section 8.7).

```
132    // display this fragment's menu items
133    @Override
134    public void onCreateOptionsMenu(Menu menu, MenuInflater inflater)
135    {
```

Fig. 8.32 | `ContactListFragment` overridden methods `onCreateOptionsMenu` and `onOptionsItemSelected`. (Part 1 of 2.)

```
      super.onCreateOptionsMenu(menu, inflater);
      inflater.inflate(R.menu.fragment_contact_list_menu, menu);
   }

   // handle choice from options menu
   @Override
   public boolean onOptionsItemSelected(MenuItem item)
   {
      switch (item.getItemId())
      {
         case R.id.action_add:
            listener.onAddContact();
            return true;
      }

      return super.onOptionsItemSelected(item); // call super's method
   }

```

Fig. 8.32 | `ContactListFragment` overridden methods `onCreateOptionsMenu` and `onOptionsItemSelected`. (Part 2 of 2.)

`ContactListFragment` Method `updateContactList`

Method `updateContactList` (Fig. 8.33) creates and executes a `GetContactsTask` to update the contact list.

```
   // update data set
   public void updateContactList()
   {
      new GetContactsTask().execute((Object[]) null);
   }
} // end class ContactListFragment
```

Fig. 8.33 | `ContactListFragment` method `updateContactList`.

8.7 `AddEditFragment` Class

The `AddEditFragment` (Figs. 8.34–8.40) provides the interface for adding new contacts or editing existing ones.

`AddEditFragment` `package` Statement and `import` Statements

Figure 8.34 lists the `package` statement and `import` statements for class `AddEditFragment`. No new classes are used in this `Fragment`.

```
// AddEditFragment.java
// Allows user to add a new contact or edit an existing one
package com.deitel.addressbook;

```

Fig. 8.34 | `AddEditFragment` `package` statement and `import` statements. (Part 1 of 2.)

```
import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.app.Fragment;
import android.content.Context;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup;
import android.view.inputmethod.InputMethodManager;
import android.widget.Button;
import android.widget.EditText;

public class AddEditFragment extends Fragment
{
```

Fig. 8.34 | `AddEditFragment` `package` statement and `import` statements. (Part 2 of 2.)

`AddEditFragmentListener` *Interface*

Figure 8.35 declares the nested interface `AddEditFragmentListener` containing the callback method `onAddEditCompleted` that `MainActivity` implements to be notified when the user saves a new contact or saves changes to an existing one.

```
   // callback method implemented by MainActivity
   public interface AddEditFragmentListener
   {
      // called after edit completed so contact can be redisplayed
      public void onAddEditCompleted(long rowID);
   }

```

Fig. 8.35 | `AddEditFragmentListener` interface.

`AddEditFragment` *Instance Variables*

Figure 8.36 lists the class's instance variables:

- Variable `listener` refers to the `AddEditFragmentListener` that's notified when the user clicks the **Save Contact** button.
- Variable `rowID` represents the current contact being manipulated if this `Fragment` was displayed to allow the user to edit an existing contact.
- Variable `contactInfoBundle` will be `null` if a new contact is being added or will refer to a `Bundle` of contact information if an existing contact is being edited.
- The instance variables at lines 36–42 will refer to the `Fragment`'s `EditText`s.

```
30    private AddEditFragmentListener listener;
31
32    private long rowID; // database row ID of the contact
33    private Bundle contactInfoBundle; // arguments for editing a contact
34
35    // EditTexts for contact information
36    private EditText nameEditText;
37    private EditText phoneEditText;
38    private EditText emailEditText;
39    private EditText streetEditText;
40    private EditText cityEditText;
41    private EditText stateEditText;
42    private EditText zipEditText;
43
```

Fig. 8.36 | `AddEditFragment` instance variables.

`AddEditFragment` Overridden Methods `onAttach` and `onDetach`

Class `AddEditFragment` overrides `Fragment` lifecycle methods `onAttach` and `onDetach` (Fig. 8.37) to set instance variable `listener` to refer to the host `Activity` (line 49) when the `AddEditFragment` is attached and to set `listener` to `null` (line 57) when the `AddEditFragment` is detached.

```
44    // set AddEditFragmentListener when Fragment attached
45    @Override
46    public void onAttach(Activity activity)
47    {
48       super.onAttach(activity);
49       listener = (AddEditFragmentListener) activity;
50    }
51
52    // remove AddEditFragmentListener when Fragment detached
53    @Override
54    public void onDetach()
55    {
56       super.onDetach();
57       listener = null;
58    }
59
```

Fig. 8.37 | AddEditFragment overridden methods `onAttach` and `onDetach`.

`AddEditFragment` Overridden Method `onCreateView`

In method `onCreateView` (Fig. 8.38), lines 70–78 inflate the GUI and get the `Fragment`'s `EditTexts`. Next, we use `Fragment` method `getArguments` to get the `Bundle` of arguments (if any). When we launch the `AddEditFragment` from the `MainActivity`, we don't pass a `Bundle`, because the user is adding a new contact's information. In this case, `getArguments` will return `null`. If it returns a `Bundle` (line 82), then the `AddEditFragment` was launched from the `DetailsFragment` and the user chose to edit an existing contact. Lines 84–91 read the arguments out of the `Bundle` by calling methods `getLong` (line 84) and `get-`

String, and the String data is displayed in the EditTexts for editing. Lines 95–97 register a listener (Fig. 8.39) for the **Save Contact** Button.

```
 60      // called when Fragment's view needs to be created
 61      @Override
 62      public View onCreateView(LayoutInflater inflater, ViewGroup container,
 63         Bundle savedInstanceState)
 64      {
 65         super.onCreateView(inflater, container, savedInstanceState);
 66         setRetainInstance(true); // save fragment across config changes
 67         setHasOptionsMenu(true); // fragment has menu items to display
 68
 69         // inflate GUI and get references to EditTexts
 70         View view =
 71            inflater.inflate(R.layout.fragment_add_edit, container, false);
 72         nameEditText = (EditText) view.findViewById(R.id.nameEditText);
 73         phoneEditText = (EditText) view.findViewById(R.id.phoneEditText);
 74         emailEditText = (EditText) view.findViewById(R.id.emailEditText);
 75         streetEditText = (EditText) view.findViewById(R.id.streetEditText);
 76         cityEditText = (EditText) view.findViewById(R.id.cityEditText);
 77         stateEditText = (EditText) view.findViewById(R.id.stateEditText);
 78         zipEditText = (EditText) view.findViewById(R.id.zipEditText);
 79
 80         contactInfoBundle = getArguments(); // null if creating new contact
 81
 82         if (contactInfoBundle != null)
 83         {
 84            rowID = contactInfoBundle.getLong(MainActivity.ROW_ID);
 85            nameEditText.setText(contactInfoBundle.getString("name"));
 86            phoneEditText.setText(contactInfoBundle.getString("phone"));
 87            emailEditText.setText(contactInfoBundle.getString("email"));
 88            streetEditText.setText(contactInfoBundle.getString("street"));
 89            cityEditText.setText(contactInfoBundle.getString("city"));
 90            stateEditText.setText(contactInfoBundle.getString("state"));
 91            zipEditText.setText(contactInfoBundle.getString("zip"));
 92         }
 93
 94         // set Save Contact Button's event listener
 95         Button saveContactButton =
 96            (Button) view.findViewById(R.id.saveContactButton);
 97         saveContactButton.setOnClickListener(saveContactButtonClicked);
 98         return view;
 99      }
100
```

Fig. 8.38 | AddEditFragment overridden method onCreateView.

OnClickListener to Process Save Contact Button *Events*

When the user touches the **Save Contact** Button, the saveContactButtonClicked listener (Fig. 8.39) executes. To save a contact, the user must enter at least the contact's name. Method onClick ensures that the length of the name is greater than 0 characters (line 107) and, if so, creates and executes an AsyncTask (to perform the save operation). Method doInBackground (lines 113–118) calls saveContact (Fig. 8.40) to save the contact into the database. Method onPostExecute (lines 120–131) programmatically hides the keyboard (lines 124–128), then notifies MainActivity that a contact was saved (line 130). If the nameEditText is empty, lines 139–153 display a DialogFragment telling the user that a contact name must be provided to save the contact.

```
101    // responds to event generated when user saves a contact
102    OnClickListener saveContactButtonClicked = new OnClickListener()
103    {
104       @Override
105       public void onClick(View v)
106       {
107          if (nameEditText.getText().toString().trim().length() != 0)
108          {
109             // AsyncTask to save contact, then notify listener
110             AsyncTask<Object, Object, Object> saveContactTask =
111                new AsyncTask<Object, Object, Object>()
112                {
113                   @Override
114                   protected Object doInBackground(Object... params)
115                   {
116                      saveContact(); // save contact to the database
117                      return null;
118                   }
119
120                   @Override
121                   protected void onPostExecute(Object result)
122                   {
123                      // hide soft keyboard
124                      InputMethodManager imm = (InputMethodManager)
125                         getActivity().getSystemService(
126                            Context.INPUT_METHOD_SERVICE);
127                      imm.hideSoftInputFromWindow(
128                         getView().getWindowToken(), 0);
129
130                      listener.onAddEditCompleted(rowID);
131                   }
132                }; // end AsyncTask
133
134             // save the contact to the database using a separate thread
135             saveContactTask.execute((Object[]) null);
136          }
137          else // required contact name is blank, so display error dialog
138          {
139             DialogFragment errorSaving =
140                new DialogFragment()
141                {
142                   @Override
143                   public Dialog onCreateDialog(Bundle savedInstanceState)
144                   {
145                      AlertDialog.Builder builder =
146                         new AlertDialog.Builder(getActivity());
147                      builder.setMessage(R.string.error_message);
```

Fig. 8.39 | OnClickListener to process **Save Contact** Button events. (Part 1 of 2.)

```
148                      builder.setPositiveButton(R.string.ok, null);
149                      return builder.create();
150                   }
151                };
152
153             errorSaving.show(getFragmentManager(), "error saving contact");
154          }
155       } // end method onClick
156    }; // end OnClickListener saveContactButtonClicked
157
```

Fig. 8.39 | `OnClickListener` to process **Save Contact** `Button` events. (Part 2 of 2.)

`AddEditFragment` Method `saveContact`

The `saveContact` method (Fig. 8.40) saves the information in this `Fragment`'s `EditText`s. First, lines 162–163 create the `DatabaseConnector` object, then we check whether the `contactInfoBundle` is `null`. If so, this is a new contact and lines 168–175 get the `String`s from the `EditText`s and pass them to the `DatabaseConnector` object's `insertContact` method to create the new contact. If the `Bundle` is not `null`, an existing contact is being updated. In this case, we get the `String`s from the `EditText`s and pass them to the `DatabaseConnector` object's `updateContact` method, using the existing `rowID` to indicate which record to update. `DatabaseConnector` methods `insertContact` and `updateContact` each handle opening and closing the database.

```
158    // saves contact information to the database
159    private void saveContact()
160    {
161       // get DatabaseConnector to interact with the SQLite database
162       DatabaseConnector databaseConnector =
163          new DatabaseConnector(getActivity());
164
165       if (contactInfoBundle == null)
166       {
167          // insert the contact information into the database
168          rowID = databaseConnector.insertContact(
169             nameEditText.getText().toString(),
170             phoneEditText.getText().toString(),
171             emailEditText.getText().toString(),
172             streetEditText.getText().toString(),
173             cityEditText.getText().toString(),
174             stateEditText.getText().toString(),
175             zipEditText.getText().toString());
176       }
177       else
178       {
179          databaseConnector.updateContact(rowID,
180             nameEditText.getText().toString(),
181             phoneEditText.getText().toString(),
182             emailEditText.getText().toString(),
```

Fig. 8.40 | `AddEditFragment` method `saveContact`. (Part 1 of 2.)

```
            streetEditText.getText().toString(),
            cityEditText.getText().toString(),
            stateEditText.getText().toString(),
            zipEditText.getText().toString());
      }
   } // end method saveContact
} // end class AddEditFragment
```

Fig. 8.40 | `AddEditFragment` method `saveContact`. (Part 2 of 2.)

8.8 `DetailsFragment` Class

The `DetailsFragment` (Figs. 8.41–8.50) displays one contact's information and provides menu items that enable the user to edit or delete that contact.

`DetailsFragment` `package` Statement and `import` Statements

Figure 8.41 lists the `package` statement, the `import` statements and the beginning of class `ContactListFragment`'s declaration. There are no new classes and interfaces used in this class.

```
// DetailsFragment.java
// Displays one contact's details
package com.deitel.addressbook;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.app.Fragment;
import android.content.DialogInterface;
import android.database.Cursor;
import android.os.AsyncTask;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class DetailsFragment extends Fragment
{
```

Fig. 8.41 | `DetailsFragment` `package` statement and `import` statements.

`DetailsFragmentListener` Interface

Figure 8.42 declares the nested interface `DetailsFragmentListener` containing the callback methods that `MainActivity` implements to be notified when the user deletes a contact (line 28) and when the user touches the edit menu item to edit a contact (line 31).

```
24     // callback methods implemented by MainActivity
25     public interface DetailsFragmentListener
26     {
27        // called when a contact is deleted
28        public void onContactDeleted();
29
30        // called to pass Bundle of contact's info for editing
31        public void onEditContact(Bundle arguments);
32     }
33
```

Fig. 8.42 | `DetailsFragmentListener` interface.

`DetailsFragment` Instance Variables

Figure 8.43 shows the class's instance variables. Line 34 declares variable `listener` which will refer to the object (`MainActivity`) that implements the `DetailsFragmentListener` interface. Variable `rowID` represents the current contact's unique row ID in the database. The `TextView` instance variables (lines 37–43) are used to display the contact's data on the screen.

```
34     private DetailsFragmentListener listener;
35
36     private long rowID = -1; // selected contact's rowID
37     private TextView nameTextView; // displays contact's name
38     private TextView phoneTextView; // displays contact's phone
39     private TextView emailTextView; // displays contact's email
40     private TextView streetTextView; // displays contact's street
41     private TextView cityTextView; // displays contact's city
42     private TextView stateTextView; // displays contact's state
43     private TextView zipTextView; // displays contact's zip
44
```

Fig. 8.43 | `DetailsFragment` instance variables.

`DetailsFragment` Overridden Methods `onAttach` and `onDetach`

Class `DetailsFragment` overrides `Fragment` lifecycle methods `onAttach` and `onDetach` (Fig. 8.44) to set instance variable `listener` when the `DetailsFragment` is attached and detached, respectively.

```
45     // set DetailsFragmentListener when fragment attached
46     @Override
47     public void onAttach(Activity activity)
48     {
49        super.onAttach(activity);
50        listener = (DetailsFragmentListener) activity;
51     }
52
```

Fig. 8.44 | `DetailsFragment` overridden methods `onAttach` and `onDetach`. (Part 1 of 2.)

```
    // remove DetailsFragmentListener when fragment detached
    @Override
    public void onDetach()
    {
       super.onDetach();
       listener = null;
    }

```

Fig. 8.44 | DetailsFragment overridden methods onAttach and onDetach. (Part 2 of 2.)

DetailsFragment Overridden Method onCreateView

The onCreateView method (Fig. 8.45) obtains the selected contact's row ID (lines 70–79). If the Fragment is being restored, we load the rowID from the savedInstanceState bundle; otherwise, we get it from the Fragment's Bundle of arguments. Lines 82–93 inflate the GUI and get references to the TextViews.

```
61     // called when DetailsFragmentListener's view needs to be created
62     @Override
63     public View onCreateView(LayoutInflater inflater, ViewGroup container,
64        Bundle savedInstanceState)
65     {
66        super.onCreateView(inflater, container, savedInstanceState);
67        setRetainInstance(true); // save fragment across config changes
68
69        // if DetailsFragment is being restored, get saved row ID
70        if (savedInstanceState != null)
71           rowID = savedInstanceState.getLong(MainActivity.ROW_ID);
72        else
73        {
74           // get Bundle of arguments then extract the contact's row ID
75           Bundle arguments = getArguments();
76
77           if (arguments != null)
78              rowID = arguments.getLong(MainActivity.ROW_ID);
79        }
80
81        // inflate DetailsFragment's layout
82        View view =
83           inflater.inflate(R.layout.fragment_details, container, false);
84        setHasOptionsMenu(true); // this fragment has menu items to display
85
86        // get the EditTexts
87        nameTextView = (TextView) view.findViewById(R.id.nameTextView);
88        phoneTextView = (TextView) view.findViewById(R.id.phoneTextView);
89        emailTextView = (TextView) view.findViewById(R.id.emailTextView);
90        streetTextView = (TextView) view.findViewById(R.id.streetTextView);
91        cityTextView = (TextView) view.findViewById(R.id.cityTextView);
92        stateTextView = (TextView) view.findViewById(R.id.stateTextView);
93        zipTextView = (TextView) view.findViewById(R.id.zipTextView);
```

Fig. 8.45 | DetailsFragment overridden method onCreateView. (Part 1 of 2.)

```
94          return view;
95       }
96
```

Fig. 8.45 | `DetailsFragment` overridden method `onCreateView`. (Part 2 of 2.)

`DetailsFragment` Overridden Method `onResume`

`Fragment` lifecycle method `onResume` (Fig. 8.46) creates and executes an `AsyncTask` (line 102) of type `LoadContactTask` (defined in Fig. 8.49) that gets the specified contact from the database and displays its data. Method `execute`'s argument in this case is the `rowID` of the contact to load. Every time line 102 executes, it creates a new `LoadContactTask` object—again, this is required because each `AsyncTask` can be executed *only once*.

```
97       // called when the DetailsFragment resumes
98       @Override
99       public void onResume()
100      {
101         super.onResume();
102         new LoadContactTask().execute(rowID); // load contact at rowID
103      }
104
```

Fig. 8.46 | `DetailsFragment` overridden method `onResume`.

`DetailsFragment` Overridden Method `onSaveInstanceState`

`Fragment` method `onSaveInstanceState` (Fig. 8.47) saves the selected contact's rowID when the configuration of the device changes during the app's execution—for example, when the user rotates the device or slides out a keyboard on a device with a hard keyboard. The state of the GUI components is saved for you automatically, but any other items that you wish to restore during a configuration change should be stored in the `Bundle` that `onSaveInstanceState` receives.

```
105      // save currently displayed contact's row ID
106      @Override
107      public void onSaveInstanceState(Bundle outState)
108      {
109          super.onSaveInstanceState(outState);
110          outState.putLong(MainActivity.ROW_ID, rowID);
111      }
112
```

Fig. 8.47 | `DetailsFragment` overridden method `onSaveInstanceState`.

`DetailsFragment` Overridden Methods `onCreateOptionsMenu` and `onOptionsItemSelected`

The `DetailsFragment`'s menu provides options for editing the current contact and for deleting it. Method `onCreateOptionsMenu` (Fig. 8.48, lines 114–119) inflates the menu resource file `fragment_details_menu.xml`. Method `onOptionsItemSelected` (lines 122–

146) uses the selected MenuItem's resource ID to determine which one was selected. If the user selected the menu item with ID R.id.action_edit, lines 129–137 create a Bundle containing the contact's data, then line 138 passes the Bundle to the DetailsFragment-Listener for use in the AddEditFragment. If the user selected the menu item with ID R.id.action_delete, line 141 calls method deleteContact (Fig. 8.50).

```
113    // display this fragment's menu items
114    @Override
115    public void onCreateOptionsMenu(Menu menu, MenuInflater inflater)
116    {
117       super.onCreateOptionsMenu(menu, inflater);
118       inflater.inflate(R.menu.fragment_details_menu, menu);
119    }
120
121    // handle menu item selections
122    @Override
123    public boolean onOptionsItemSelected(MenuItem item)
124    {
125       switch (item.getItemId())
126       {
127          case R.id.action_edit:
128             // create Bundle containing contact data to edit
129             Bundle arguments = new Bundle();
130             arguments.putLong(MainActivity.ROW_ID, rowID);
131             arguments.putCharSequence("name", nameTextView.getText());
132             arguments.putCharSequence("phone", phoneTextView.getText());
133             arguments.putCharSequence("email", emailTextView.getText());
134             arguments.putCharSequence("street", streetTextView.getText());
135             arguments.putCharSequence("city", cityTextView.getText());
136             arguments.putCharSequence("state", stateTextView.getText());
137             arguments.putCharSequence("zip", zipTextView.getText());
138             listener.onEditContact(arguments); // pass Bundle to listener
139             return true;
140          case R.id.action_delete:
141             deleteContact();
142             return true;
143       }
144
145       return super.onOptionsItemSelected(item);
146    }
147
```

Fig. 8.48 | DetailsFragment overridden methods onCreateOptionsMenu and onOptionsItemSelected.

LoadContactTask Subclass of AsyncTask

Nested class LoadContactTask (Fig. 8.49) extends class AsyncTask and defines how to interact with the database to get one contact's information for display. In this case the three generic type parameters are:

- Long for the variable-length argument list passed to AsyncTask's doInBackground method. This will contain the row ID needed to locate one contact.

- Object for the variable-length argument list passed to AsyncTask's onProgress-Update method, which we don't use in this example.
- Cursor for the type of the task's result, which is passed to the AsyncTask's on-PostExecute method.

```
   // performs database query outside GUI thread
   private class LoadContactTask extends AsyncTask<Long, Object, Cursor>
   {
      DatabaseConnector databaseConnector =
         new DatabaseConnector(getActivity());

      // open database & get Cursor representing specified contact's data
      @Override
      protected Cursor doInBackground(Long... params)
      {
         databaseConnector.open();
         return databaseConnector.getOneContact(params[0]);
      }

      // use the Cursor returned from the doInBackground method
      @Override
      protected void onPostExecute(Cursor result)
      {
         super.onPostExecute(result);
         result.moveToFirst(); // move to the first item

         // get the column index for each data item
         int nameIndex = result.getColumnIndex("name");
         int phoneIndex = result.getColumnIndex("phone");
         int emailIndex = result.getColumnIndex("email");
         int streetIndex = result.getColumnIndex("street");
         int cityIndex = result.getColumnIndex("city");
         int stateIndex = result.getColumnIndex("state");
         int zipIndex = result.getColumnIndex("zip");

         // fill TextViews with the retrieved data
         nameTextView.setText(result.getString(nameIndex));
         phoneTextView.setText(result.getString(phoneIndex));
         emailTextView.setText(result.getString(emailIndex));
         streetTextView.setText(result.getString(streetIndex));
         cityTextView.setText(result.getString(cityIndex));
         stateTextView.setText(result.getString(stateIndex));
         zipTextView.setText(result.getString(zipIndex));

         result.close(); // close the result cursor
         databaseConnector.close(); // close database connection
      } // end method onPostExecute
   } // end class LoadContactTask

```

Fig. 8.49 | LoadContactTask subclass of AsyncTask.

Lines 151–152 create a new object of our DatabaseConnector class (Section 8.9). Method doInBackground (lines 155–160) opens the connection to the database and calls the DatabaseConnector's getOneContact method, which queries the database to get the contact with the specified rowID that was passed as the only argument to this AsyncTask's execute method. In doInBackground, the rowID is stored in params[0].

The resulting Cursor is passed to method onPostExecute (lines 163–189). The Cursor is positioned *before* the first row of the result set. In this case, the result set will contain only one record, so Cursor method **moveToFirst** (line 167) can be used to move the Cursor to the first row in the result set. [*Note:* It's considered good practice to ensure that Cursor method moveToFirst returns true before attempting to get data from the Cursor. In this app, there will always be a row in the Cursor.]

We use Cursor's **getColumnIndex method** (lines 170–176) to get the column indices for the columns in the database's contacts table. (We hard coded the column names in this app, but these could be implemented as String constants as we did for ROW_ID in class MainActivity in Fig. 8.14.) This method returns -1 if the column is not in the query result. Class Cursor also provides method **getColumnIndexOrThrow** if you prefer to get an exception when the specified column name does not exist. Lines 179–185 use Cursor's **getString method** to retrieve the String values from the Cursor's columns, then display these values in the corresponding TextViews. Lines 187–188 close the Cursor and the connection to the database, as they're no longer needed. It's good practice to release resources like database connections when they are not being used so that other activities can use the resources.

Method **deleteContact** *and DialogFragment* **confirmDelete**

Method deleteContact (Fig. 8.50, lines 193–197) displays a DialogFragment (lines 200–252) asking the user to confirm that the currently displayed contact should be deleted. If so, the DialogFragment uses an AsyncTask to delete the contact from the database. If the user clicks the **Delete** Button in the dialog, lines 222–223 create a new DatabaseConnector. Lines 226–241 create an AsyncTask that, when executed (line 244), passes a Long value representing the contact's row ID to the doInBackground, which then deletes the contact. Line 232 calls the DatabaseConnector's deleteContact method to perform the actual deletion. When the doInBackground completes execution, line 239 calls the listener's onContactDeleted method so that MainActivity can remove the DetailsFragment from the screen.

```
192    // delete a contact
193    private void deleteContact()
194    {
195       // use FragmentManager to display the confirmDelete DialogFragment
196       confirmDelete.show(getFragmentManager(), "confirm delete");
197    }
198
199    // DialogFragment to confirm deletion of contact
200    private DialogFragment confirmDelete =
201       new DialogFragment()
202       {
```

Fig. 8.50 | Method deleteContact and DialogFragment confirmDelete. (Part 1 of 2.)

```
         // create an AlertDialog and return it
         @Override
         public Dialog onCreateDialog(Bundle bundle)
         {
            // create a new AlertDialog Builder
            AlertDialog.Builder builder =
               new AlertDialog.Builder(getActivity());

            builder.setTitle(R.string.confirm_title);
            builder.setMessage(R.string.confirm_message);

            // provide an OK button that simply dismisses the dialog
            builder.setPositiveButton(R.string.button_delete,
               new DialogInterface.OnClickListener()
               {
                  @Override
                  public void onClick(
                     DialogInterface dialog, int button)
                  {
                     final DatabaseConnector databaseConnector =
                        new DatabaseConnector(getActivity());

                     // AsyncTask deletes contact and notifies listener
                     AsyncTask<Long, Object, Object> deleteTask =
                        new AsyncTask<Long, Object, Object>()
                        {
                           @Override
                           protected Object doInBackground(Long... params)
                           {
                              databaseConnector.deleteContact(params[0]);
                              return null;
                           }

                           @Override
                           protected void onPostExecute(Object result)
                           {
                              listener.onContactDeleted();
                           }
                        }; // end new AsyncTask

                     // execute the AsyncTask to delete contact at rowID
                    deleteTask.execute(new Long[] { rowID });
                  } // end method onClick
               } // end anonymous inner class
            ); // end call to method setPositiveButton

            builder.setNegativeButton(R.string.button_cancel, null);
            return builder.create(); // return the AlertDialog
         }
      }; // end DialogFragment anonymous inner class
} // end class DetailsFragment
```

Fig. 8.50 | Method `deleteContact` and `DialogFragment` `confirmDelete`. (Part 2 of 2.)

8.9 DatabaseConnector Utility Class

The `DatabaseConnector` utility class (Figs. 8.51–8.58) manages this app's interactions with SQLite for creating and manipulating the `UserContacts` database, which contains one table named `contacts`.

`package` Statement, `import` Statements and Fields

Figure 8.51 lists class `DatabaseConnector`'s `package` statement, `import` statements and fields. We've highlighted the `import` statements for the new classes and interfaces discussed in Section 8.3. The `String` constant `DATABASE_NAME` (line 16) specifies the name of the database that will be created or opened. *Database names must be unique within a specific app but need not be unique across apps.* A `SQLiteDatabase` object (line 18) provides read/write access to a SQLite database. The `DatabaseOpenHelper` (line 19) is a `private` nested class that extends abstract class `SQLiteOpenHelper`—such a class is used to manage creating, opening and upgrading databases (perhaps to modify a database's structure). We discuss `SQLiteOpenHelper` in more detail in Fig. 8.58.

```
 1  // DatabaseConnector.java
 2  // Provides easy connection and creation of UserContacts database.
 3  package com.deitel.addressbook;
 4
 5  import android.content.ContentValues;
 6  import android.content.Context;
 7  import android.database.Cursor;
 8  import android.database.SQLException;
 9  import android.database.sqlite.SQLiteDatabase;
10  import android.database.sqlite.SQLiteOpenHelper;
11  import android.database.sqlite.SQLiteDatabase.CursorFactory;
12
13  public class DatabaseConnector
14  {
15     // database name
16     private static final String DATABASE_NAME = "UserContacts";
17
18     private SQLiteDatabase database; // for interacting with the database
19     private DatabaseOpenHelper databaseOpenHelper; // creates the database
20
```

Fig. 8.51 | `DatabaseConnector` class's `package` statement, `import` statements and instance variables.

`DatabaseConnector` Constructor and Methods `open` and `close`

`DatabaseConnection`'s constructor (Fig. 8.52, lines 22–27) creates a new object of class `DatabaseOpenHelper` (Fig. 8.58), which will be used to open or create the database. We discuss the details of the `DatabaseOpenHelper` constructor in Fig. 8.58. The `open` method (lines 30–34) attempts to establish a connection to the database and throws a `SQLException` if the connection attempt fails. Method **`getWritableDatabase`** (line 33), which is inherited from `SQLiteOpenHelper`, returns a `SQLiteDatabase` object. If the database has not yet been created, this method will create it; otherwise, the method will open it. Once the database is opened successfully, it will be *cached* by the operating system to improve the performance of future database interactions. The `close` method (lines 37–41) closes the database connection by calling the inherited `SQLiteOpenHelper` method **`close`**.

```
21      // public constructor for DatabaseConnector
22      public DatabaseConnector(Context context)
23      {
24         // create a new DatabaseOpenHelper
25         databaseOpenHelper =
26            new DatabaseOpenHelper(context, DATABASE_NAME, null, 1);
27      }
28
29      // open the database connection
30      public void open() throws SQLException
31      {
32         // create or open a database for reading/writing
33         database = databaseOpenHelper.getWritableDatabase();
34      }
35
36      // close the database connection
37      public void close()
38      {
39         if (database != null)
40            database.close(); // close the database connection
41      }
42
```

Fig. 8.52 | `DatabaseConnector` constructor and methods `open` and `close`.

`DatabaseConnector` Method `insertContact`

Method `insertContact` (Fig. 8.53) inserts a new contact with the given information into the database. We first put each piece of contact information into a new **`ContentValues`** object (lines 47–54), which maintains a map of key–value pairs—the database's column names are the keys. Lines 56–58 open the database, insert the new contact and close the database. `SQLiteDatabase`'s **`insert` method** (line 57) inserts the values from the given `ContentValues` into the table specified as the first argument—the `"contacts"` table in this case. The second parameter of this method, which is not used in this app, is named `nullColumnHack` and is needed because *SQLite does not support inserting a completely empty row into a table*—this would be the equivalent of passing an empty `ContentValues` object to `insert`. Instead of making it illegal to pass an empty `ContentValues` to the method, the `nullColumnHack` parameter is used to identify a column that accepts `NULL` values.

```
43      // inserts a new contact in the database
44      public long insertContact(String name, String phone, String email,
45         String street, String city, String state, String zip)
46      {
47         ContentValues newContact = new ContentValues();
48         newContact.put("name", name);
49         newContact.put("phone", phone);
```

Fig. 8.53 | `DatabaseConnector` method `insertContact`. (Part 1 of 2.)

```
50         newContact.put("email", email);
51         newContact.put("street", street);
52         newContact.put("city", city);
53         newContact.put("state", state);
54         newContact.put("zip", zip);
55
56         open(); // open the database
57         long rowID = database.insert("contacts", null, newContact);
58         close(); // close the database
59         return rowID;
60      } // end method insertContact
61
```

Fig. 8.53 | `DatabaseConnector` method `insertContact`. (Part 2 of 2.)

`DatabaseConnector` *Method* ***`updateContact`***

Method `updateContact` (Fig. 8.54) is similar to method `insertContact`, except that it calls `SQLiteDatabase`'s **`update` method** (line 76) to update an existing contact. The `update` method's third argument represents a SQL `WHERE` clause (without the keyword `WHERE`) that specifies which record(s) to update. In this case, we use the record's row ID to update a specific contact.

```
62      // updates an existing contact in the database
63      public void updateContact(long id, String name, String phone,
64         String email, String street, String city, String state, String zip)
65      {
66         ContentValues editContact = new ContentValues();
67         editContact.put("name", name);
68         editContact.put("phone", phone);
69         editContact.put("email", email);
70         editContact.put("street", street);
71         editContact.put("city", city);
72         editContact.put("state", state);
73         editContact.put("zip", zip);
74
75         open(); // open the database
76         database.update("contacts", editContact, "_id=" + id, null);
77         close(); // close the database
78      }
79
```

Fig. 8.54 | `DatabaseConnector` method `updateContact`.

Method ***`getAllContacts`***

Method `getAllContacts` (Fig. 8.55) uses `SqLiteDatabase`'s **`query` method** (lines 83–84) to retrieve a `Cursor` that provides access to the IDs and names of all the contacts in the database. The arguments are:

- the name of the table to query.
- a `String` array of the column names to return (the `_id` and `name` columns here)—`null` returns all columns in the table, which is generally a poor programming

practice, because to conserve memory, processor time and battery power, you should obtain only the data you need.

- a SQL WHERE clause (without the keyword WHERE), or null to return all rows.
- a String array of arguments to be substituted into the WHERE clause wherever ? is used as a placeholder for an argument value, or null if there are no arguments in the WHERE clause.
- a SQL GROUP BY clause (without the keywords GROUP BY), or null if you don't want to group the results.
- a SQL HAVING clause (without the keyword HAVING) to specify which groups from the GROUP BY clause to include in the results—null is required if the GROUP BY clause is null.
- a SQL ORDER BY clause (without the keywords ORDER BY) to specify the order of the results, or null if you don't wish to specify the order.

The Cursor returned by method query contains all the table rows that match the method's arguments—the so-called *result set*. The Cursor is positioned *before* the first row of the result set—Cursor's various move methods can be used to move the Cursor through the result set for processing.

```
   // return a Cursor with all contact names in the database
   public Cursor getAllContacts()
   {
      return database.query("contacts", new String[] {"_id",
         null, null, null, null, "name");
   }

```

Fig. 8.55 | DatabaseConnector method getAllContacts.

Method getOneContact

Method getOneContact (Fig. 8.56) also uses SqLiteDatabase's query method to query the database. In this case, we retrieve all the columns in the database for the contact with the specified ID.

```
   // return a Cursor containing specified contact's information
   public Cursor getOneContact(long id)
   {
      return database.query(
         "contacts", null, "_id=" + id, null, null, null, null);
   }

```

Fig. 8.56 | DatabaseConnector method getOneContact.

Method deleteContact

Method deleteContact (Fig. 8.57) uses SqLiteDatabase's **delete method** (line 98) to delete a contact from the database. In this case, we retrieve all the columns in the database for the contact with the specified ID. The three arguments are the database table from which to delete the record, the `WHERE` clause (without the keyword `WHERE`) and, if the `WHERE` clause has arguments, a `String` array of values to substitute into the `WHERE` clause (`null` in our case).

```
   // delete the contact specified by the given String name
   public void deleteContact(long id)
   {
      open(); // open the database
      database.delete("contacts", "_id=" + id, null);
      close(); // close the database
   }

```

Fig. 8.57 | `DatabaseConnector` method `deleteContact`.

`private` Nested Class `DatabaseOpenHelper` That Extends `SQLiteOpenHelper`

The `private` nested class `DatabaseOpenHelper` (Fig. 8.58) extends abstract class `SQLite-OpenHelper`, which helps apps create databases and manage version changes. The constructor (lines 105–109) simply calls the superclass constructor, which requires four arguments:

- the `Context` in which the database is being created or opened,
- the database name—this can be `null` if you wish to use an in-memory database,
- the `CursorFactory` to use—`null` indicates that you wish to use the default SQLite `CursorFactory` (typically for most apps) and
- the database version number (starting from 1).

You must override this class's abstract methods `onCreate` and `onUpgrade`. If the database does not yet exist, the `DatabaseOpenHelper`'s **`onCreate` method** will be called to create it. If you supply a newer version number than the database version currently stored on the device, the `DatabaseOpenHelper`'s **`onUpgrade` method** will be called to upgrade the database to the new version (perhaps to add tables or to add columns to an existing table).

```
102   private class DatabaseOpenHelper extends SQLiteOpenHelper
103   {
104      // constructor
105      public DatabaseOpenHelper(Context context, String name,
106         CursorFactory factory, int version)
107      {
108         super(context, name, factory, version);
109      }
110
111      // creates the contacts table when the database is created
112      @Override
113      public void onCreate(SQLiteDatabase db)
114      {
```

Fig. 8.58 | `SQLiteOpenHelper` class `DatabaseOpenHelper`. (Part 1 of 2.)

```
115           // query to create a new table named contacts
116           String createQuery = "CREATE TABLE contacts" +
117              "(_id integer primary key autoincrement," +
118              "name TEXT, phone TEXT, email TEXT, " +
119              "street TEXT, city TEXT, state TEXT, zip TEXT);";
120
121           db.execSQL(createQuery); // execute query to create the database
122        }
123
124        @Override
125        public void onUpgrade(SQLiteDatabase db, int oldVersion,
126           int newVersion)
127        {
128        }
129     } // end class DatabaseOpenHelper
130  } // end class DatabaseConnector
```

Fig. 8.58 | `SQLiteOpenHelper` class `DatabaseOpenHelper`. (Part 2 of 2.)

The `onCreate` method (lines 112–122) specifies the table to create with the SQL `CREATE TABLE` command, which is defined as a `String` (lines 116–119). In this case, the contacts table contains an integer primary key field (`_id`) that's auto-incremented, and text fields for all the other columns. Line 121 uses `SQLiteDatabase`'s **`execSQL`** method to execute the `CREATE TABLE` command. Since we don't need to upgrade the database, we simply override method `onUpgrade` with an empty body. Class `SQLiteOpenHelper` also provides the **`onDowngrade` method** that can be used to downgrade a database when the currently stored version has a higher version number than the one requested in the call to class `SQLiteOpenHelper`'s constructor. Downgrading might be used to revert the database back to a prior version with fewer columns in a table or fewer tables in the database—perhaps to fix a bug in the app.

All the `SQLiteDatabase` methods we used in class `DatabaseConnector` have corresponding methods which perform the same operations but throw exceptions on failure, as opposed to simply returning `-1` (e.g., `insertOrThrow` vs. `insert`). These methods are interchangeable, allowing you to decide how to deal with database read and write errors.

8.10 Wrap-Up

In this chapter, you created an **Address Book** app that enables users to add, view, edit and delete contact information that's stored in a SQLite database. You defined common GUI component attribute–value pairs as XML `style` resources, then applied the styles to all components that share those values by using the components' `style` attribute. You added a border to a `TextView` by specifying a `Drawable` as the value for the `TextView`'s `android:background` attribute and you created a custom `Drawable` using an XML representation of a `shape`. You also used Android standard icons to enhance the visual appearance of the app's menu items.

When an `Fragment`'s primary task is to display a scrollable list of items, you learned that you can extend class `ListFragment` to create a `Fragment` that displays a `ListView` in its default layout. You used this to display the contacts stored in the app's database. You bound data to the `ListView` via a `CursorAdapter` that displayed the results of a database query.

In this app's `Activity`, you used `FragmentTransactions` to add `Fragments` to and replace `Fragments` in the GUI dynamically. You also used the `Fragment` back stack to support the back button for returning to a previously displayed `Fragment` and to allow the app's `Activity` to programmatically return to previous `Fragments`.

We demonstrated how to communicate data between `Fragments` and a host `Activity` or the `Activity`'s other `Fragments` via interfaces of callback methods that are implemented by the host `Activity`. You also used `Bundles` to pass arguments to `Fragments`.

You used a subclass of `SQLiteOpenHelper` to simplify creating the database and to obtain a `SQLiteDatabase` object for manipulating a database's contents. You processed query results via a `Cursor`. You used subclasses of `AsyncTask` to perform database tasks outside the GUI thread and return results to the GUI thread. This allowed you to take advantage of Android's threading capabilities without directly creating and manipulating threads.

In Chapter 9, we discuss the business side of Android app development. You'll see how to prepare your app for submission to Google Play, including making icons. We'll discuss how to test your apps on devices and publish them on Google Play. We discuss the characteristics of great apps and the Android design guidelines to follow. We provide tips for pricing and marketing your app. We also review the benefits of offering your app for free to drive sales of other products, such as a more feature-rich version of the app or premium content. We show how to use Google Play to track app sales, payments and more.

Self-Review Exercises

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

a) __________ method is called by the system when the configuration of the device changes during the app's execution.

b) To get a database operation's results in the GUI thread, you use a(n) __________ (package `android.os`) to perform the operation in one thread and receive the results in the GUI thread.

c) `Fragment` method __________ returns the `Bundle` of arguments to the `Fragment`.

d) The `Cursor` returned by method `query` contains all the table rows that match the method's arguments—the so-called __________.

8.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) It's good practice to release resources like database connections when they are not being used so that other activities can use the resources.

b) It's considered good practice to ensure that `Cursor` method `moveToFirst` returns `false` before attempting to get data from the `Cursor`.

c) It's good practice to perform long-running operations or operations that block execution until they complete (e.g., file and database access) in the GUI thread.

d) `SimpleCursorAdapter` is a subclass of `CursorAdapter` that's designed to simplify mapping `Cursor` columns directly to `TextViews` or `ImagesViews` defined in your XML layouts.

e) A key benefit of using an `SyncTask` is that it handles the details of creating threads and executing its methods on the appropriate threads for you, so that you do not have to interact with the threading mechanism directly.

Answers to Self-Review Exercises

8.1 a) `onSaveInstanceState`. b) `AsynkTask`. c) `getArguments`. d) result set

8.2 a) True. b) False. It's considered good practice to ensure that `Cursor` method `moveToFirst` returns `true` before attempting to get data from the `Cursor`. c) False. It's good practice to perform long-running operations or operations that block execution until they complete (e.g., file and database access) *outside* the GUI thread. d) True. e) False. A key benefit of using an `AsyncTask` is that it handles the details of creating threads and executing its methods on the appropriate threads for you, so that you do not have to interact with the threading mechanism directly.

Exercises

8.3 *(Flag Quiz App Modification)* Revise the **Flag Quiz** app to use one `Activity`, dynamic `Fragments` and `FragmentTransactions` as you did in the **Address Book** app.

8.4 *(Coin Collection App)* Using the techniques you learned in this chapter, create an app that allows you to enter information about your coin collection. Provide fields for the currency value, country, year, and any other fields you'd like to track. The app should provide similar activities to the `Address Book` app for viewing the list of coins (in alphabetical order of their country), adding and/or updating the information for a coin and viewing the details of a coin.

8.5 *(Recipe App)* Using the techniques you learned in this chapter, create a cooking recipe app. Provide fields for the recipe name, category (e.g., appetizer, entree, desert, salad, side dish), a list of the ingredients and instructions for preparing the dish. The app should provide similar activities to the **Address Book** app for viewing the list of recipes (in alphabetical order), adding and/or updating a recipe and viewing the details of a recipe.

8.6 *(Favorite Twitter Searches App Enhancement)* Using the techniques you learned in this chapter, modify the **Favorite Twitter Searches** app so that it loads and saves the `SharedPreferences` in a separate thread of execution.

8.7 *(Grocery Shop App)* Create an app that allows the user to enter, edit and save a list of groceries. Include a favorites feature that allows the user to easily add additional groceries. Include an optional feature to input a price for each grocery item and a quantity so that the user can track the total cost of all items on the list.

8.8 *(Expense Tracker App)* Create an app that allows the user to keep track of personal expenses. Provide categories for classifying each expense (e.g., monthly expenses, travel, entertainment, necessities). Provide an option for tagging recurring expenses that automatically adds the expense to a calendar at the proper frequency (daily, weekly, monthly or yearly). Optional: Investigate Android's status-bar notifications mechanism at `developer.android.com/guide/topics/ui/notifiers/index.html`. Provide notifications to remind the user when a bill is due.

8.9 *(Cooking with Healthier Ingredients App)* Obesity in the United States is increasing at an alarming rate. Check the map from the Centers for Disease Control and Prevention (CDC) at `www.cdc.gov/obesity/data/adult.html`, which shows obesity trends in the United States over the last 20 years. As obesity increases, so do occurrences of related problems (e.g., heart disease, high blood pressure, high cholesterol, type 2 diabetes). Create an app that helps users choose healthier ingredients when cooking, and helps those allergic to certain foods (e.g., nuts, gluten) find substitutes. The app should allow the user to enter a recipe, then should suggest healthier replacements for some of the ingredients. For simplicity, your app should assume the recipe has no abbreviations for measures such as teaspoons, cups, and tablespoons, and uses numerical digits for quantities (e.g., 1 egg, 2 cups) rather than spelling them out (one egg, two cups). Some common substitutions are shown in Fig. 8.59. Your app should display a warning such as, "Always consult your physician before making significant changes to your diet."

Ingredient	Substitution
1 cup sour cream	1 cup yogurt
1 cup milk	1/2 cup evaporated milk and 1/2 cup water
1 teaspoon lemon juice	1/2 teaspoon vinegar
1 cup sugar	1/2 cup honey, 1 cup molasses or 1/4 cup agave nectar
1 cup butter	1 cup margarine or yogurt
1 cup flour	1 cup rye or rice flour
1 cup mayonnaise	1 cup cottage cheese or 1/8 cup mayonnaise and 7/8 cup yogurt
1 egg	2 tablespoons cornstarch, arrowroot flour or potato starch or 2 egg whites or 1/2 large banana (mashed)
1 cup milk	1 cup soy milk
1/4 cup oil	1/4 cup applesauce
white bread	whole-grain bread
1 cup sour cream	1 cup yogurt

Fig. 8.59 | Common ingredient substitutions.

The app should take into consideration that replacements are not always one-for-one. For example, if a cake recipe calls for three eggs, it might reasonably use six egg whites instead. Conversion data for measurements and substitutes can be obtained at websites such as:

```
http://chinesefood.about.com/od/recipeconversionfaqs/f/usmetricrecipes.htm
http://www.pioneerthinking.com/eggsub.html
http://www.gourmetsleuth.com/conversions.htm
```

Your app should consider the user's health concerns, such as high cholesterol, high blood pressure, weight loss, gluten allergy, and so on. For high cholesterol, the app should suggest substitutes for eggs and dairy products; if the user wishes to lose weight, low-calorie substitutes for ingredients such as sugar should be suggested.

8.10 *(Crossword Puzzle Generator App)* Most people have worked a crossword puzzle, but few have ever attempted to generate one. Create a personal crossword generator app that allows the user to enter words and corresponding hints. Once the user completes this task, generate a crossword puzzle using the supplied words. Display the corresponding hints when the user touches the first square in a word. If the square represents the beginning of both a horizontal and vertical word, show both hints.

9 Google Play and App Business Issues

Objectives

In this chapter you'll be introduced to:

- Preparing your apps for publication.
- Pricing your apps and the benefits of free vs. paid apps.
- Monetizing your apps with in-app advertising.
- Selling virtual goods using in-app billing.
- Registering for Google Play.
- Setting up a Google Wallet merchant account.
- Uploading your apps to Google Play.
- Launching the **Play Store** from within an app.
- Other Android app marketplaces.
- Other popular mobile app platforms to which you can port your apps to broaden your market.
- Marketing your apps.

Outline

9.1 Introduction

In Chapters 2–8, we developed a variety of complete working Android apps. Once you've developed and tested your own apps, both in the emulator and on Android devices, the next step is to submit them to Google Play—and/or other app marketplaces—for distribution to a worldwide audience. In this chapter, you'll learn how to register for Google Play and set up a Google Wallet account so that you can sell your apps. You'll learn how to prepare your apps for publication and how to upload them to Google Play. In a few cases, we'll refer you to Android documentation instead of showing the steps in the book, because the steps are likely to change. We'll tell you about additional Android app marketplaces where you can distribute your apps. We'll discuss whether you should offer your apps for free or for a fee, and mention key resources for monetizing apps such as in-app advertising and selling virtual goods. We'll provide resources for marketing your apps, and mention other app platforms to which you may port your Android apps to broaden your marketplace.

9.2 Preparing Your Apps for Publication

The *Preparing for Release* section in the *Dev Guide* (`http://developer.android.com/tools/publishing/preparing.html`) lists items to consider before publishing your app on Google Play, including:

- *Testing* your app on Android devices
- Including an *End User License Agreement* with your app (optional)
- Adding an *icon* and label to the app's manifest
- *Versioning* your app (e.g., 1.0, 1.1, 2.0, 2.3, 3.0)
- Getting a *cryptographic key* for *digitally signing* your app
- *Compiling* your app

You should also read the *Launch Checklist* (`http://developer.android.com/distribute/googleplay/publish/preparing.html`) and the *Tablet App Quality Checklist* (`http://developer.android.com/distribute/googleplay/quality/tablet.html`) before publishing your app.

9.2.1 Testing Your App

Before submitting your app to Google Play, test it thoroughly on a variety of devices. Although the app might work perfectly using the emulator on your computer, problems could arise when running it on particular Android devices. The Google Play Developer Console now provides support for alpha and beta testing apps with groups of people through Google+. For more information, visit:

```
https://play.google.com/apps/publish/
```

9.2.2 End User License Agreement

You have the option to include an **End User License Agreement (EULA)** with your app. An EULA is an agreement through which you license your software to the user. It typically stipulates terms of use, limitations on redistribution and reverse engineering, product liability, compliance with applicable laws and more. You might want to consult an attorney when drafting an EULA for your app. To view a sample EULA, see

```
http://www.rocketlawyer.com/document/end-user-license-agreement.rl
```

9.2.3 Icons and Labels

Design an icon for your app and provide a text label (a name) that will appear in Google Play and on the user's device. The icon could be your company logo, an image from the app or a custom image. The Android Asset Studio provides a tool for creating app icons:

```
http://android-ui-utils.googlecode.com/hg/asset-studio/dist/
index.html
```

Create a version of your icon for each of these screen densities:

- `xx-high` (XXHDPI): 144 x 144 pixels
- `x-high` (XHDPI): 96 x 96 pixels
- `high` (HDPI): 72 x 72 pixels
- `medium` (MDPI): 48 x 48 pixels

You'll also need a high-resolution icon for use in Google Play. This icon should be:

- 512 x 512 pixels
- 32-bit PNG
- 1 MB maximum

Since the icon is the most important brand asset, having one that's high quality is important. Consider hiring an experienced graphic designer to help you create a compelling, professional icon. Figure 9.1 lists several design firms that offer free, professionally designed icons and paid custom icon design services. Once you've created the icon and label, you'll need to specify them in the app's `AndroidManifest.xml` file by setting the `android:icon` and `android:label` attributes of the application element.

Company	URL	Services
glyphlab	`http://www.glyphlab.com/icon_design/`	Custom icon design and some *free* downloadable icons.
Androidicons	`http://www.androidicons.com`	Designs custom icons, sells a set of 200 icons for a flat fee and has some *free* downloadable icons.
Iconiza	`http://www.iconiza.com`	Designs custom icons for a flat fee and sells stock icons.
Aha-Soft	`http://www.aha-soft.com/icon-design.htm`	Designs custom icons for a flat fee.
Rosetta®	`http://icondesign.rosetta.com/`	Designs custom icons for a fee.
Elance®	`http://www.elance.com`	Search for freelance icon designers.

Fig. 9.1 | Custom app icon design firms.

9.2.4 Versioning Your App

It's important to include a *version name* (shown to the users) and a *version code* (an integer version number used internally by Google Play) for your app, and to consider your strategy for numbering updates. For example, the first version name of your app might be 1.0, minor updates might be 1.1 and 1.2, and the next major update might be 2.0. The version code is an integer that typically starts at 1 and is incremented by 1 for each new version of your app that you post. For additional guidelines, see *Versioning Your Applications* at

```
http://developer.android.com/tools/publishing/versioning.html
```

9.2.5 Licensing to Control Access to Paid Apps

The Google Play ***licensing service*** allows you to create licensing policies to control access to your paid apps. For example, you might use a licensing policy to limit how many simultaneous device installs are allowed. To learn more about the licensing service, visit

```
http://developer.android.com/google/play/licensing/index.html
```

9.2.6 Obfuscating Your Code

You should "obfuscate" any apps you upload to Google Play to discourage reverse engineering of your code and further protect your apps. The free **ProGuard** tool—which runs when you build your app in *release mode*—shrinks the size of your `.apk` file (the Android app package file that contains your app for installation) and optimizes and obfuscates the code "by removing unused code and renaming classes, fields, and methods with semantically obscure names."[1] To learn how to set up and use the ProGuard tool, go to

```
http://developer.android.com/tools/help/proguard.html
```

1. `http://developer.android.com/tools/help/proguard.html#enabling`.

For additional information about protecting your apps from piracy using code obfuscation, visit

```
http://www.techrepublic.com/blog/app-builder/
   protect-your-android-apps-with-obfuscation/1724
```

9.2.7 Getting a Private Key for Digitally Signing Your App

Before uploading your app to a device, Google Play or other app marketplaces, you must *digitally sign* the `.apk` file using a **digital certificate** that identifies you as the author of the app. A digital certificate includes your name or company name, contact information, etc. It can be self-signed using a **private key** (i.e., a secure password used to *encrypt* the certificate); you do not need to purchase a certificate from a third-party certificate authority (though it's an option). Eclipse automatically digitally signs your app when you execute it in an emulator or on a device for *debugging* purposes. That digital certificate is *not* valid for use with Google Play, and it expires 365 days after it's created. For detailed instructions on digitally signing your apps, see *Signing Your Applications* at:

```
http://developer.android.com/tools/publishing/app-signing.html
```

9.2.8 Screenshots

Take *at least* two screenshots of your app (you may upload a maximum of eight screenshots each for a smartphone, a 7" tablet and a 10" tablet) that will be included with your app description in Google Play (Fig. 9.2). These provide a preview of your app, since users can't test the app before downloading it (although they may return an app for a refund within 15 minutes after purchasing and downloading it). Choose attractive screenshots that show the app's functionality.

Specification	Description
Size	Minimum dimension of 320 pixels and maximum dimension of 3840 pixels (the maximum dimension may not be more than twice the length of the minimum).
Format	24-bit PNG or JPEG format with no alpha (transparency) effects.
Image	Full bleed to the edge with no borders.

Fig. 9.2 | Screenshot specifications.

The Dalvik Debug Monitor Service (DDMS), which is installed with the ADT Plugin for Eclipse and helps you debug your apps running on actual devices, also enables you to capture screenshots on your device. To do so, perform the following steps:

1. Run the app on your device as described at the end of Section 1.9.
2. In Eclipse, select **Window > Open Perspective > DDMS**, which allows you to use the DDMS tools.
3. In the **Devices** window (Fig. 9.3), select the device from which you'd like to obtain a screen capture.

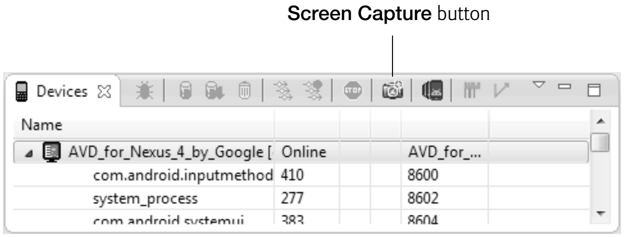

Screen Capture button
Devices
Name
AVD_for_Nexus_4_by_Google [Online AVD_for_...
com.android.inputmethod 410 8600
system_process 277 8602
com.android.systemui 383 8604

Fig. 9.3 | **Devices** window in the DDMS perspective.

4. Click the **Screen Capture** button to display the **Device Screen Capture** window (Fig. 9.4).

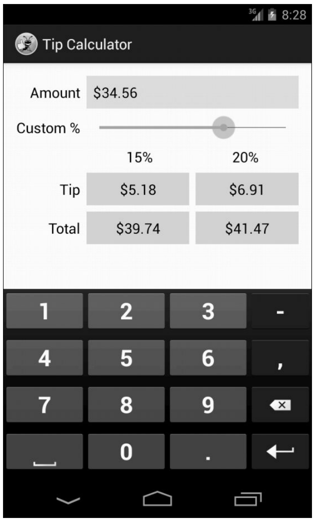


Fig. 9.4 | **Device Screen Capture** window showing a capture of the **Tip Calculator** app from Chapter 3.

5. After you've ensured that the screen is showing what you'd like to capture, click the **Save** button to save the image.
6. If you wish to change what's on your device's screen before saving the image, make the change on the device, then press the **Refresh** button in the **Device Screen Capture** window to recapture the device's screen.

9.2.9 Promotional App Video

When you upload your app to Google Play, you'll have the option to include a URL for a short promotional video on YouTube. Figure 9.5 lists several examples. Some videos show a person holding a device and interacting with the app. Other videos use screen captures. Figure 9.6 lists several video creation tools and services (some free, some paid).

App	URL
Temple Run®: Oz	`http://www.youtube.com/watch?v=QM9sT1ydtj0`
GT Racing: Motor Academy	`http://www.youtube.com/watch?v=2Z9OPICdgoA`
Beach Buggy Blitz™	`http://www.youtube.com/watch?v=YqDczawTsYw`
Real Estate and Homes by Trulia®	`http://www.youtube.com/watch?v=rLn697AszGs`
Zappos.com®	`http://www.youtube.com/watch?v=U-oNyK9kl_Q`
Megopolis International	`http://www.youtube.com/watch?v=JrqeEJ1xzCY`

Fig. 9.5 | Examples of promotional videos for apps in Google Play.

Tools and services	URL
Animoto	`http://animoto.com`
Apptamin	`http://www.apptamin.com`
Movie Maker for Microsoft Windows	`http://windows.microsoft.com/en-us/windows-live/movie-maker`
CamStudio™	`http://camstudio.org`
Jing	`http://www.techsmith.com/jing.html`
Camtasia Studio®	`http://www.techsmith.com/camtasia.html`
TurboDemo™	`http://www.turbodemo.com/eng/index.php`

Fig. 9.6 | Tools and services for creating promotional videos.

To upload your video, create an account or sign into your existing YouTube account. Click **Upload** at the top-right of the page. Click **Select files to upload** to choose a video from your computer or simply drag and drop the video file onto the web page.

9.3 Pricing Your App: Free or Fee

You set the prices for your apps that are distributed through Google Play. Many developers offer their apps for free as a marketing, publicity and branding tool, earning revenue through increased sales of products and services, sales of more feature-rich versions of the same apps and sales of additional content through the apps using *in-app purchase* or *in-app advertising*. Figure 9.7 lists ways to monetize your apps.

Ways to monetize an app
• Sell the app in Google Play.
• Sell the app in other Android app marketplaces.

Fig. 9.7 | Ways to monetize an app. (Part 1 of 2.)

Ways to monetize an app

- Sell paid upgrades.
- Sell virtual goods (Section 9.5).
- Sell an app to a company that brands it as their own.
- Use mobile advertising services for in-app ads (Section 9.4).
- Sell in-app advertising space directly to your customers.
- Use it to drive sales of a more feature-rich version of the app.

Fig. 9.7 | Ways to monetize an app. (Part 2 of 2.)

9.3.1 Paid Apps

The average price for apps varies widely by category. For example, according to the app discovery site AppBrain (`http://www.appbrain.com`), the average price for puzzle game apps is $1.54 and for business apps is $6.47.[2] Although these prices may seem low, keep in mind that successful apps could sell tens of thousands, hundreds of thousands or even millions of copies.

When setting a price for your app, start by researching your competition. How much do they charge? Do their apps have similar functionality? Is yours more feature-rich? Will offering your app at a lower price than the competition attract users? Is your goal to recoup development costs and generate additional revenue?

If you change your strategy, you can eventually offer your paid app for free permantently. However it's not currently possible to change your free apps to paid.

Financial transactions for paid apps in Google Play are handled by Google Wallet (`http://google.com/wallet`), though customers of some mobile carriers (such as AT&T, Sprint and T-Mobile) can opt to use carrier billing to charge paid apps to their wireless bill. Earnings are paid to Google Wallet merchants monthly.[3] You're responsible for paying taxes on the revenue you earn through Google Play.

9.3.2 Free Apps

Approximately 80% of apps on Google Play are free, and they comprise the vast majority of downloads.[4] Given that users are more likely to download an app if it's free, consider offering a free "lite" version of your app to encourage users to try it. For example, if your app is a game, you might offer a free lite version with just the first few levels. When the user has finished playing the free levels, the app would offer an option to buy through Google Play your more robust app with numerous game levels. Or, your app would display a message that the user can purchase additional levels from within the app for a more seamless upgrade (see Section 9.5). According to a recent study by AdMob, *upgrading from the "lite" version is the number-one reason why users purchase a paid app.*[5]

2. `http://www.appbrain.com/stats/android-market-app-categories`.
3. `http://support.google.com/googleplay/android-developer/answer/137997?hl=en&ref_topic=15867`.
4. `http://www.gartner.com/newsroom/id/2592315`.
5. `http://metrics.admob.com/wp-content/uploads/2009/08/AdMob-Mobile-Metrics-July-09.pdf`.

Many companies use free apps to build brand awareness and drive sales of other products and services (Fig. 9.8).

Free app	Functionality
Amazon® Mobile	Browse and purchase items on Amazon.
Bank of America	Locate ATMs and bank branches in your area, check balances and pay bills.
Best Buy®	Browse and purchase items.
CNN	Get the latest world news, receive breaking news alerts and watch live video.
Epicurious Recipe	View thousands of recipes from several Condé Nast magazines, including *Gourmet* and *Bon Appetit*.
ESPN® ScoreCenter	Set up personalized scoreboards to track your favorite college and professional sports teams.
NFL Mobile	Get the latest NFL news and updates, live programming, NFL Replay and more.
UPS® Mobile	Track shipments, find drop-off locations, get estimated shipping costs and more.
NYTimes	Read articles from the *New York Times*, free of charge.
Pocket Agent™	State Farm Insurance's app enables you contact an agent, file claims, find local repair centers, check your State Farm bank and mutual fund accounts and more.
Progressive® Insurance	Report a claim and submit photos from the scene of a car accident, find a local agent, get car safety information when you're shopping for a new car and more.
USA Today®	Read articles from *USA Today* and get the latest sports scores.
Wells Fargo® Mobile	Locate ATMs and bank branches in your area, check balances, make transfers and pay bills.
Women's Health Workouts Lite	View numerous workouts from one of the leading women's magazines.

Fig. 9.8 | Companies using free Android apps to build brand awareness.

9.4 Monetizing Apps with In-App Advertising

Many developers offer free apps monetized with **in-app advertising**—often banner ads similar to those you find on websites. Mobile advertising networks such as AdMob (`http://www.admob.com/`) and Google AdSense for Mobile (`http://www.google.com/mobileads/publisher_home.html`) aggregate advertisers for you and serve relevant ads to your app (see Section 9.13). You earn advertising revenue based on the number of click-throughs. The top 100 free apps might earn a few hundred dollars to a few thousand dollars per day. In-app advertising does not generate significant revenue for most apps, so if your goal is to recoup development costs and generate profits, you should consider charging a fee for your app.

9.5 Monetizing Apps: Using In-App Billing to Sell Virtual Goods

Google Play's **in-app billing** service (`http://developer.android.com/google/play/billing/index.html`) enables you to sell **virtual goods** (e.g., digital content) through apps on devices running Android 2.3 or higher (Fig. 9.9). According to Google, apps that use in-app billing earn profoundly more revenue than paid apps alone. Of the 24 top-grossing apps on Google Play, 23 use in-app billing.[6] The in-app billing service is available only for apps purchased through Google Play; it may *not* be used in apps sold through third-party app stores. To use in-app billing, you'll need a Google Play publisher account (see Section 9.6) and a Google Wallet merchant account (see Section 9.7). Google pays you 70% of the revenue for all in-app purchases made through your apps.

Virtual goods		
Magazine e-subscriptions	Localized guides	Avatars
Virtual apparel	Additional game levels	Game scenery
Add-on features	Ringtones	Icons
E-cards	E-gifts	Virtual currency
Wallpapers	Images	Virtual pets
Audios	Videos	E-books and more.

Fig. 9.9 | Virtual goods.

Selling virtual goods can generate higher revenue *per user* than in-app advertising.[7] A few apps that have been particularly successful selling virtual goods include Angry Birds, DragonVale, Zynga Poker, Bejeweled Blitz, NYTimes and Candy Crush Saga. Virtual goods are particularly popular in mobile games.

To implement in-app billing, follow the steps at

```
http://developer.android.com/google/play/billing/
billing_integrate.html
```

For additional information about in-app billing, including subscriptions, sample apps, security best practices, testing and more, visit `http://developer.android.com/google/play/billing/billing_overview.html`. You can also take the free *Selling In-app Products* training class at

```
http://developer.android.com/training/in-app-billing/index.html
```

In-App Purchase for Apps Sold Through Other App Marketplaces

If you choose to sell your apps through other marketplaces (see Section 9.11), several third-party mobile payment providers can enable you to build *in-app purchase* into your apps using

6. `http://android-developers.blogspot.com/2012/05/in-app-subscriptions-in-google-play.html`.
7. `http://www.businessinsider.com/its-morning-in-venture-capital-2012-5?utm_source=readme&utm_medium=righttrail&utm_term=&utm_content=6&utm_campaign=recirc`.

APIs from mobile payment providers (Fig. 9.10)—you cannot use Google Play's in-app billing. Start by building the additional *locked functionality* (e.g., game levels, avatars) into your app. When the user opts to make a purchase, the in-app purchasing tool handles the financial transaction and returns a message to the app verifying payment. The app then unlocks the additional functionality. Mobile carriers collect between 25% and 45% of the price.

Provider	URL	Description
PayPal Mobile Payments Library	`http://developer.paypal.com/webapps/developer/docs/classic/mobile/gs_MPL/`	Users click the **Pay with PayPal** button, log into their PayPal account, then click **Pay.**
Amazon In-App Purchasing	`http://developer.amazon.com/sdk/in-app-purchasing.html`	In-app purchase for apps sold through the Amazon App Store for Android.
Zong	`http://www.zong.com/android`	Provides **Buy** button for one-click payment. Payments appear on the user's phone bill.
Samsung In-App Purchase	`http://developer.samsung.com/android/tools-sdks/In-App-Purchase-Library`	In-app purchase for apps designed specifically for Samsung devices.
Boku	`http://www.boku.com`	Users click **Pay by Mobile**, enter their mobile phone number, then complete the transaction by replying to a text message sent to their phone.

Fig. 9.10 | Mobile payment providers for in-app purchase.

9.6 Registering at Google Play

To publish your apps on Google Play, you must register for an account at

```
http://play.google.com/apps/publish
```

There's a one-time $25 registration fee. Unlike other popular mobile platforms, *Google Play has no approval process for uploading apps*. You must, however, adhere to the *Google Play Developer Program Policies*. If your app is in violation of these policies, it can be removed at any time; serious or repeated violations may result in account termination (Fig. 9.11).

Violations of the *Google Play Content Policy for Developers*	
• Infringing on others' intellectual property rights (e.g., trademarks, patents and copyrights).	• Invading personal privacy.
• Illegal activities.	• Interfering with the services of other parties.
	• Harming the user's device or personal data.
	• Gambling.

Fig. 9.11 | Some violations of the *Google Play Content Policy for Developers* (`http://play.google.com/about/developer-content-policy.html#showlanguages`). (Part 1 of 2.)

Violations of the *Google Play Content Policy for Developers*	
• Creating a "spammy" user experience (e.g., misleading the user about the app's purpose).	• Promoting hate or violence.
• Adversely impacting a user's service charges or a wireless carrier's network.	• Providing pornographic or obscene content, or anything unsuitable for children under age 18.
• Impersonation or deception.	• Ads in system-level notifications and widgets.

Fig. 9.11 | Some violations of the *Google Play Content Policy for Developers* (`http://play.google.com/about/developer-content-policy.html#showlanguages`). (Part 2 of 2.)

9.7 Setting Up a Google Wallet Merchant Account

To sell your apps on Google Play, you'll need a **Google Wallet merchant account**, available to Google Play developers in 32 countries (Fig. 9.12).[8] Google Wallet is used as a payment service for online transactions. Once you've registered and logged into Google Play at `http://play.google.com/apps/publish/`, click the **Financial Reports** link, then click **Set up a merchant account**. You'll need to

- provide private information by which Google can contact you
- provide customer-support contact information where users can contact you
- provide financial information so that Google may perform a credit check
- agree to the Terms of Service, which describe the features of the service, permissible transactions, prohibited actions, service fees, payment terms and more.

Countries			
Argentina	Denmark	Italy	Russia
Australia	France	Mexico	Spain
Austria	Germany	Netherlands	South Korea
Belgium	Hong Kong	New Zealand	Sweden
Brazil	India	Norway	Switzerland
Canada	Ireland	Poland	Taiwan
Czech Republic	Israel	Portugal	United Kingdom
Finland	Japan	Singapore	United States

Fig. 9.12 | Countries in which Google Wallet merchant accounts are available.

Google Wallet processes payments and helps protect you from fraudulent purchases. The standard payment processing rates are waived for your Google Play sales.[9] Google pays you 70% of the app price. Once you set up a Google Wallet account, you'll be able

8. `http://support.google.com/googleplay/android-developer/answer/150324?hl=en&ref_topic=15867`.
9. `http://checkout.google.com/termsOfService?type=SELLER`.

to use it for more activities than just selling your apps, such as making purchases at participating stores.

9.8 Uploading Your Apps to Google Play

Once you've prepared your files and you're ready to upload your app, review the steps in the *Launch Checklist* at:

```
http://developer.android.com/distribute/googleplay/publish/
   preparing.html
```

Then log into Google Play at `http://play.google.com/apps/publish` (Section 9.6) and click the **Publish an Android App on Google Play** button to begin the upload process. You will be asked to upload the following assets:

1. *App* `.apk` *file* that includes the app's code files, assets, resources and the manifest file.
2. At least *two screenshots* of your app to be included in Google Play. You may include screenshots for an Android phone, 7" tablet and 10" tablet.
3. *High-resolution app icon (*512 x 512 pixels) to be included in Google Play.
4. *Promotional graphic* (optional) for Google Play to be used by Google if they decide to promote your app (for examples, check out some of the graphics for featured apps on Google Play). The graphic must be 180 pixels wide by 120 pixels tall in 24-bit PNG or JPEG format with *no alpha transparency effects*. It must also have a full bleed (i.e., go to the edge of the screen with no border in the graphic).
5. *Promotional video* (optional) to be included in Google Play. You may include a URL for a promotional video for your app (e.g., a YouTube link to a video that demonstrates how your app works).

In addition to app assets, you will be asked to provide the following additional listing details for Google Play:

1. ***Language.*** By default, your app will be listed in English. If you'd like to list it in additional languages, select them from the list provided (Fig. 9.13).

Language				
Afrikaans	Amharic	Arabic	Belarusian	Catalan
Chinese (simplified or traditional)		Croatian	Czech	Danish
Dutch	English (UK or United States)		Estonian	Filipino
Finnish	French	German	Greek	Hebrew
Hindi	Hungarian	Indonesian	Italian	Japanese
Korean	Latvian	Lithuanian	Malay	Norwegian
Persian	Polish	Portuguese (Brazil or Portugal)		Romanian
Romansh	Russian	Serbian	Slovak	Slovenian
Spanish (Latin America, Spain or United States)			Swahili	Swedish
Thai	Turkish	Ukrainian	Vietnamese	Zulu

Fig. 9.13 | Languages for listing apps in Google Play.

2. *Title.* The title of your app as it will appear in Google Play (30 characters maximum). It does *not* need to be unique among all Android apps.
3. *Description.* A description of your app and its features (4,000 characters maximum). It's recommended that you use the last portion of the description to explain why each permission is required and how it's used.
4. *Recent changes.* A walkthrough of any changes specific to the latest version of your app (500 characters maximum).
5. *Promo text.* The promotional text for marketing your app (80 characters max).
6. *App type.* Choose **Applications** or **Games**.
7. *Category.* Select the category (see Fig. 1.8) that best suits your game or app.
8. *Price.* The default setting is **Free**. To sell your app for a fee, you'll need to set up a merchant account at Google Wallet.
9. *Content rating.* You may select **High Maturity**, **Medium Maturity**, **Low Maturity** or **Everyone**. For more information, see *Rating your application content for Google Play* at `http://support.google.com/googleplay/android-developer/answer/188189`.
10. *Locations.* By default, the app will be listed in all current and future Google Play countries. If you do not want your app to be available in all these countries, you may pick and choose specific ones where you'd like your app to be listed.
11. *Website.* A **Visit Developer's Website** link will be included in your app's listing in Google Play. Provide a direct link to the page on your website where users interested in downloading your app can find more information, including marketing copy, feature listings, additional screenshots, instructions, etc.
12. *E-mail.* Your e-mail address will also be included in Google Play, so that customers can contact you with questions, report errors, etc.
13. *Phone number.* Sometimes your phone number is included in Google Play. Therefore it's recommended that you leave this field blank unless you provide phone support. You may want to provide a customer service phone number on your website.

9.9 Launching the Play Store from Within Your App

To drive additional sales of your apps, you can launch the **Play Store** app (Google Play) from within your app (typically by including a button) so that the user can download other apps you've published or purchase a related app with functionality beyond that of the previously downloaded "lite" version. You can also launch the **Play Store** app to enable users to download the latest updates.

There are two ways to launch the **Play Store** app. First, you can bring up Google Play search results for apps with a specific developer name, package name or string of characters. For example, if you want to encourage users to download other apps you've published, you could include a button in your app that, when touched, launches the **Play Store** app and initiates a search for apps containing your name or company name. The second option is to bring the user to the details page in the **Play Store** app for a specific app.

To learn about launching **Play Store** from within an app, see *Linking Your Products* at `http://developer.android.com/distribute/googleplay/promote/linking.html`.

9.10 Managing Your Apps in Google Play

The *Google Play Developer Console* allows you to manage your account and your apps, check users' star ratings for your apps (0 to 5 stars), respond to users' comments, track the overall number of installs of each app and the number of active installs (installs minus uninstalls). You can view installation trends and the distribution of app downloads across Android versions, devices, and more. Crash reports list any crash and freeze information from users. If you've made upgrades to your app, you can easily publish the new version. You can remove the app from Google Play, but users who downloaded it previously may keep it on their devices. Users who uninstalled the app will be able to reinstall it even after it's been removed (it will remain on Google's servers unless it's removed for violating the Terms of Service).

9.11 Other Android App Marketplaces

In addition to Google Play, you may choose to make your apps available through other Android app marketplaces (Fig. 9.14), or through your own website using services such as AndroidLicenser (`http://www.androidlicenser.com`). To learn more about releasing your app through a website see

```
http://developer.android.com/tools/publishing/
publishing_overview.html
```

Marketplace	URL
Amazon Appstore	`http://developer.amazon.com/welcome.html`
Opera Mobile Store	`http://apps.opera.com/en_us/index.php`
Moborobo	`http://www.moborobo.com`
Appitalism®	`http://www.appitalism.com/index.html`
Samsung Apps	`http://apps.samsung.com/mars/main/getMain.as`
GetJar	`http://www.getjar.com`
SlideMe	`http://www.slideme.org`
Handango	`http://www.handango.com`
Mplayit™	`http://www.mplayit.com`
AndroidPIT	`http://www.androidpit.com`

Fig. 9.14 | Other Android app marketplaces.

9.12 Other Popular Mobile App Platforms

According to ABI Research, 56 billion smartphone apps and 14 billion tablet apps will be downloaded in 2013.[10] By porting your Android apps to other mobile app platforms, especially to iOS (for iPhone, iPad and iPod Touch devices), you could reach an even bigger audience (Fig. 9.15). Android can be developed on Windows, Linux or Mac computers

10. `http://www.abiresearch.com/press/android-will-account-for-58-of-smartphone-app-down.`

with Java—one of the world's most widely used programming languages. However, iOS apps must be developed on Macs, which can be costly, and with the Objective-C programming language, which only a small percentage of developers know. Google has created the open-source J2ObjC tool to help you translate your Java app code to Objective-C for iOS apps. To learn more, see `http://code.google.com/p/j2objc/`.

Platform	URL	Worldwide app downloads market share
Android	`http://developer.android.com`	58% smartphone apps 17% tablet apps
iOS (Apple)	`http://developer.apple.com/ios`	33% smartphone apps 75% tablet apps
Windows Phone 8	`http://developer.windowsphone.com`	4% smartphone apps 2% tablet apps
BlackBerry (RIM)	`http://developer.blackberry.com`	3% smartphone apps
Amazon Kindle	`http://developer.amazon.com`	4% tablet apps

Fig. 9.15 | Popular mobile app platforms. (`http://www.abiresearch.com/press/android-will-account-for-58-of-smartphone-app-down`).

9.13 Marketing Your Apps

Once your app has been published, you'll want to market it to your audience.[11] *Viral marketing* through social media sites such as Facebook, Twitter, Google+ and YouTube can help you get your message out. These sites have tremendous visibility. According to a Pew Research Center study, 72% of adults on the Internet use social networks—and 67% of those are on Facebook.[12] Figure 9.16 lists some of the most popular social media sites. Also, e-mail and electronic newsletters are still effective and often inexpensive marketing tools.

Name	URL	Description
Facebook	`http://www.facebook.com`	Social networking
Twitter	`http://www.twitter.com`	Microblogging, social networking
Google+	`http://plus.google.com`	Social networking
Groupon	`http://www.groupon.com`	Daily deals

Fig. 9.16 | Popular social media sites. (Part 1 of 2.)

11. To learn more about marketing your Android apps, check out the book *Android Apps Marketing: Secrets to Selling Your Android App* by Jeffrey Hughes.
12. `http://pewinternet.org/Commentary/2012/March/Pew-Internet-Social-Networking-full-detail.aspx`.

Name	URL	Description
Foursquare	`http://www.foursquare.com`	Check-in
Pinterest	`http://www.pinterest.com`	Online pinboard
YouTube	`http://www.youtube.com`	Video sharing
LinkedIn	`http://www.linkedin.com`	Social networking for business
Flickr	`http://www.flickr.com`	Photo sharing

Fig. 9.16 | Popular social media sites. (Part 2 of 2.)

Facebook
Facebook, the premier social networking site, has more than one billion active users[13] and over 150 billion friend connections.[14] It's an excellent resource for *viral marketing*. Start by setting up an official Facebook page for your app or business. Use the page to post app information, news, updates, reviews, tips, videos, screenshots, high scores for games, user feedback and links to Google Play where users can download your app. For example, we post news and updates about Deitel publications on our Facebook page at `http://www.facebook.com/DeitelFan`.

Next, you need to spread the word. Encourage your co-workers and friends to "like" your Facebook page and ask their friends to do so as well. As people interact with your page, stories will appear in their friends' news feeds, building awareness to a growing audience.

Twitter
Twitter is a microblogging, social networking site with over 554 million active registered users.[15] You post tweets—messages of 140 characters or less. Twitter then distributes your tweets to all of your followers (at the time of this writing, one famous pop star had over 40 million followers). Many people use Twitter to track news and trends. Tweet about your app—include announcements about new releases, tips, facts, comments from users, etc. Also, encourage your colleagues and friends to tweet about your app. Use a *hashtag* (#) to reference your app. For example, when tweeting about *Android How to Program, 2/e* on our `@deitel` Twitter feed, we use the hashtag `#AndroidHTP2`. Others may use this hashtag as well to write comments about the book. This enables you to easily search tweets for messages related to the book.

Viral Video
Viral video—shared on video sites (e.g., YouTube, Bing Videos, Yahoo! Video), on social networking sites (e.g., Facebook, Twitter and Google+), through e-mail, etc.—is another great way to spread the word about your app. If you create a compelling video, perhaps one that's humorous or even outrageous, it may quickly rise in popularity and may be tagged by users across multiple social networks.

13. `http://investor.fb.com/releasedetail.cfm?ReleaseID=761090`.
14. `http://expandedramblings.com/index.php/by-the-numbers-17-amazing-facebook-stats/`.
15. `http://www.statisticbrain.com/twitter-statistics/`.

E-Mail Newsletters

If you have an e-mail newsletter, use it to promote your app. Include links to Google Play, where users can download the app. Also include links to your social networking pages, where users can stay up-to-date with the latest news about your app.

App Reviews

Contact influential bloggers and app review sites (Fig. 9.17) and tell them about your app. Provide them with a promotional code to download your app for free (see Section 9.3). Influential bloggers and reviewers receive many requests, so keep yours concise and informative without too much marketing hype. Many app reviewers post video app reviews on YouTube and other sites (Fig. 9.18).

Android app review site	URL
Android Tapp™	`http://www.androidtapp.com`
Appolicious™	`http://www.androidapps.com`
AppBrain	`http://www.appbrain.com`
AndroidZoom	`http://www.androidzoom.com`
Appstorm	http://android.appstorm.net
Best Android Apps Review	`http://www.bestandroidappsreview.com`
Android App Review Source	`http://www.androidappreviewsource.com`
Androinica	`http://www.androinica.com`
AndroidLib	`http://www.androlib.com`
Android and Me	`http://www.androidandme.com`
AndroidGuys	`http://www.androidguys.com/category/reviews`
Android Police	`http://www.androidpolice.com`
AndroidPIT	http://www.androidpit.com
Phandroid	`http://www.phandroid.com`

Fig. 9.17 | Android app review sites.

Android app review video site	URL
Daily App Show	`http://dailyappshow.com`
Crazy Mike's Apps	`http://crazymikesapps.com`
Appolicious™	`http://www.appvee.com/?device_filter=android`
Life of Android™	`http://www.lifeofandroid.com/video/`
Android Video Review	`http://www.androidvideoreview.net/`

Fig. 9.18 | Android app review video sites.

Internet Public Relations

The public relations industry uses media outlets to help companies get their message out to consumers. With the phenomenon known as Web 2.0, public relations practitioners are incorporating blogs, tweets, podcasts, RSS feeds and social media into their PR campaigns. Figure 9.19 lists some free and fee-based Internet public relations resources, including press-release distribution sites, press-release writing services and more.

Internet public relations resource	URL	Description
Free Services		
PRWeb®	`http://www.prweb.com`	Online press-release distribution service with *free* and *fee-based* services.
ClickPress™	`http://www.clickpress.com`	Submit news stories for approval (*free* of charge). If approved, they'll be available on the ClickPress site and to news search engines. For a *fee*, ClickPress will distribute your press releases globally to top financial newswires.
PRLog	`http://www.prlog.org/pub/`	*Free* press-release submission and distribution.
i-Newswire	`http://www.i-newswire.com`	*Free* and *fee-based* press-release submission and distribution.
openPR®	`http://www.openpr.com`	*Free* press-release publication.
Fee-Based Services		
PR Leap	`http://www.prleap.com`	Online press-release distribution service.
Marketwire	`http://www.marketwire.com`	Press-release distribution service allows you to target your audience by geography, industry, etc.
Mobility PR	`http://www.mobilitypr.com`	Public relations services for companies in the mobile industry.
Press Release Writing	`http://www.press-release-writing.com`	Press-release distribution and services including press-release writing, proofreading and editing. Check out the tips for writing effective press releases.

Fig. 9.19 | Internet public relations resources.

Mobile Advertising Networks

Purchasing advertising spots (e.g., in other apps, online, in newspapers and magazines or on radio and television) is another way to market your app. Mobile advertising networks (Fig. 9.20) specialize in advertising Android (and other) mobile apps on mobile platforms. Many of these networks can target audiences by location, wireless carrier, platform (e.g., Android, iOS, Windows, BlackBerry) and more. Most apps don't make much money, so be careful how much you spend on advertising.

Mobile ad networks	URL
AdMob (by Google)	`http://www.admob.com/`
Medialets	`http://www.medialets.com`
Tapjoy®	`http://www.tapjoy.com`
Nexage™	`http://www.nexage.com`
Jumptap®	`http://www.jumptap.com`
Smaato®	`http://www.smaato.com`
mMedia™	`http://mmedia.com`
InMobi™	`http://www.inmobi.com`
Flurry™	`http://www.flurry.com`

Fig. 9.20 | Mobile advertising networks.

You can also use mobile advertising networks to monetize your free apps by including ads (e.g., banners, videos) in your apps. The average eCPM (effective cost per 1,000 impressions) for ads in Android apps is $0.88, according to Opera's *State of Mobile Advertising* report[16] (though the average may vary by network, device, etc.). Most ads on Android pay based on *click-through rate (CTR)* of the ads rather than the number of impressions generated. According to a report by Jumptap, CTRs average 0.65% on mobile in-app ads,[17] though this varies based on the app, the device, targeting of the ads by the ad network and more. If your app has a lot of users and the CTRs of the ads in your apps are high, you may earn substantial advertising revenue. Also, your ad network may serve you higher-paying ads, thus increasing your earnings.

9.14 Wrap-Up

In this chapter, we walked through the process of registering for Google Play and setting up a Google Wallet account so you can sell your apps. We showed how to prepare apps for submission to Google Play, including testing them on the emulator and on Android devices, creating icons and labels, and editing the `AndroidManifest.xml` file. We walked through the steps for uploading your apps to Google Play. We showed you alternative Android app marketplaces where you can sell your apps. We provided tips for pricing your apps, and resources for monetizing them with in-app advertising and in-app sales of virtual goods. And we included resources for marketing your apps, once they're available through Google Play.

Staying in Contact with Deitel & Associates, Inc.

We hope you enjoyed reading *Android How to Program, 2/e* as much as we enjoyed writing it. We'd appreciate your feedback. Please send your questions, comments, suggestions and

16. `http://www.insidemobileapps.com/2012/12/14/ios-leads-the-pack-in-ecpm-traffic-and-revenue-on-operas-mobile-ad-platform-ipad-average-ecpm-of-4-42/`.
17. `http://paidcontent.org/2012/01/05/419-jumptap-android-the-most-popular-but-ios-still-more-interactive-for-ads/`.

corrections to `deitel@deitel.com`. Check out our growing list of Android-related Resource Centers at `http://www.deitel.com/ResourceCenters.html`. To stay up to date with the latest news about Deitel publications and corporate training sign up for the free weekly *Deitel® Buzz Online* e-mail newsletter at

```
http://www.deitel.com/newsletter/subscribe.html
```

and follow us on

- Facebook—http://`www.facebook.com/DeitelFan`
- Twitter—`@deitel`
- Google+—`http://google.com/+DeitelFan`
- YouTube—`http://youtube.com/DeitelTV`
- LinkedIn—`http://linkedin.com/company/deitel-&-associates`

Self-Review Exercises

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

a) To sell your apps on Google Play, you'll need a(n) __________ merchant account.

b) Before uploading your app to a device, to Google Play or to other app marketplaces, you must digitally sign the `.apk` file (Android app package file) using a(n) __________ that identifies you as the author of the app.

c) Installed with the ADT Plugin for Eclipse, the __________ Service helps you debug your apps running on actual devices and also enables you to capture screenshots on your device or emulator.

9.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) When an app works perfectly using the emulator on your computer, it will run on your Android device.

b) You might use a licensing policy to limit how often the app checks in with the server, how many simultaneous device installs are allowed, and what happens when an unlicensed app is identified.

c) The title of your app as it will appear in Google Play must be unique among all Android apps.

d) According to a study by app store analytics firm Distimo (`www.distimo.com/`), the average price of paid Android game apps is around $36.20.

e) According to Google, apps that use in-app billing earn profoundly more revenue than paid apps alone.

f) If you choose to sell your apps through other app marketplaces, several third-party mobile payment providers can enable you to build in-app purchase into your apps using APIs from mobile payment providers.

Answers to Self-Review Exercises

9.1 a) Google Wallet. b) digital certificate. c) Dalvik Debug Monitor.

9.2 a) False. Although the app might work perfectly using the emulator on your computer, problems could arise when running it on a particular Android device. b) True. c) False. The title of your app as it will appear in Google Play does *not* need to be unique among all Android apps.

d) False. According to the study, the the average price of game apps is around $3.27 (the median is around $2.72). e) True. f) True.

Exercises

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

a) The Google Play __________ allows you to create licensing policies to control access to your paid apps.

b) A(n) __________ is an agreement through which you license your software to the user. It typically stipulates terms of use, limitations on redistribution and reverse engineering, product liability, compliance with applicable laws and more.

c) Google Play's __________ service enables you to sell virtual goods (e.g., digital content) through apps on devices running Android 2.3 or higher.

d) According to a recent study by AdMob, __________ is the number one reason why users purchase a paid app.

e) The Google Play __________ allows you to limit how many simultaneous device installs are allowed.

9.4 State whether each of the following is *true* or *false*. If *false*, explain why.

a) You should "obfuscate" any apps you upload to Google Play to encourage reverse engineering of your code.

b) There are more paid apps than free apps on Google Play, and they comprise the vast majority of downloads.

c) Your app's version name is shown to the users, and the version code is an integer version number used internally by Google Play.

d) Free apps need not be digitally signed before uploading them to Google Play.

e) Financial transactions for paid apps in Google Play are handled by Google Wallet.

A Introduction to Java Applications

Objectives

In this appendix you'll learn:

- To write simple Java applications.
- To use input and output statements.
- Java's primitive types.
- Basic memory concepts.
- To use arithmetic operators.
- The precedence of arithmetic operators.
- To write decision-making statements.
- To use relational and equality operators.

Outline

A.1 Introduction

This appendix introduces Java application programming. You'll use tools from the JDK to compile and run programs. We've posted a Dive Into® video at `www.deitel.com/books/AndroidHTP2/` to help you get started with the popular Eclipse integrated development environment (IDE)—the most widely used Java IDE and the one that's typically used for Android app development.

A.2 Your First Program in Java: Printing a Line of Text

A Java **application** is a computer program that executes when you use the **`java` command** to launch the Java Virtual Machine (JVM).First we consider a simple application that displays a line of text. Figure A.1 shows the program followed by a box that displays its output.

```
1  // Fig. A.1: Welcome1.java
2  // Text-printing program.
3
4  public class Welcome1
5  {
6     // main method begins execution of Java application
7     public static void main( String[] args )
8     {
9        System.out.println( "Welcome to Java Programming!" );
10    } // end method main
11 } // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. A.1 | Text-printing program.

Commenting Your Programs

We insert **comments** to **document programs** and improve their readability. The Java compiler ignores comments, so they do *not* cause the computer to perform any action when the program is run.

The comment in line 1

```
// Fig. A.1: Welcome1.java
```

begins with `//`, indicating that it is an **end-of-line comment**—it terminates at the end of the line on which the `//` appears. Line 2 is a comment that describes the purpose of the program.

Java also has **traditional comments**, which can be spread over several lines as in

```
/* This is a traditional comment. It
   can be split over multiple lines */
```

These begin and end with delimiters, `/*` and `*/`. The compiler ignores all text between the delimiters.

Common Programming Error A.1

*A **syntax error** occurs when the compiler encounters code that violates Java's language rules (i.e., its syntax). Syntax errors are also called **compilation errors**, because the compiler detects them during the compilation phase. The compiler responds by issuing an error message and preventing your program from compiling.*

Using Blank Lines

Line 3 is a blank line. Blank lines, space characters and tabs make programs easier to read. Together, they're known as **white space** (or whitespace). The compiler ignores white space.

Declaring a Class

Line 4 begins a **class declaration** for class `Welcome1`. Every Java program consists of at least one class that you (the programmer) define. The **`class` keyword** introduces a class declaration and is immediately followed by the **class name** (`Welcome1`). **Keywords** are reserved for use by Java and are always spelled with all lowercase letters. The complete list of keywords can be viewed at:

```
http://bit.ly/JavaKeywords
```

Class Names and Identifiers

By convention, class names begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`). A class name is an **identifier**—a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and does not contain spaces. The name `7button` is not a valid identifier because it begins with a digit, and the name `input field` is not a valid identifier because it contains a space. Java is **case sensitive**—uppercase and lowercase letters are distinct—so `value` and `Value` are different identifiers.

In Appendices A–E, every class we define begins with the keyword **`public`**. For our application, the file name is `Welcome1.java`.

Common Programming Error A.2

A `public` class must be placed in a file that has the same name as the class (in terms of both spelling and capitalization) plus the `.java` extension; otherwise, a compilation error occurs. For example, `public` class `Welcome` must be placed in a file named `Welcome.java`.

A **left brace** (as in line 5), **`{`**, begins the **body** of every class declaration. A corresponding **right brace**, **`}`**, must end each class declaration.

Good Programming Practice A.1

Indent the entire body of each class declaration one "level" between the left brace and the right brace that delimit the body of the class. We recommend using three spaces to form a level of indent. This format emphasizes the class declaration's structure and makes it easier to read.

Declaring a Method

Line 6 is an end-of-line comment indicating the purpose of lines 7–10 of the program. Line 7 is the starting point of every Java application. The **parentheses** after the identifier `main` indicate that it's a program building block called a **method**. For a Java application, one of the methods *must* be called `main` and must be defined as shown in line 7. Methods perform tasks and can return information when they complete their tasks. Keyword **`void`** indicates that this method will *not* return any information. In line 7, the `String[] args` in parentheses is a required part of the method `main`'s declaration—we discuss this in Appendix E.

The left brace in line 8 begins the **body of the method declaration**. A corresponding right brace must end it (line 10).

Performing Output with `System.out.println`

Line 9 instructs the computer to perform an action—namely, to print the **string** of characters contained between the double quotation marks (but not the quotation marks themselves). A string is sometimes called a **character string** or a **string literal**. White-space characters in strings are *not* ignored by the compiler. Strings cannot span multiple lines of code.

The **`System.out`** object is known as the **standard output object**. It allows a Java applications to display information in the **command window** from which it executes. In recent versions of Microsoft Windows, the command window is the **Command Prompt**. In UNIX/Linux/Mac OS X, the command window is called a **terminal window** or a **shell**. Many programmers call it simply the **command line**.

Method **`System.out.println`** displays (or prints) a line of text in the command window. The string in the parentheses in line 9 is the **argument** to the method. When `System.out.println` completes its task, it positions the cursor (the location where the next character will be displayed) at the beginning of the next line in the command window.

The entire line 9, including `System.out.println`, the argument `"Welcome to Java Programming!"` in the parentheses and the **semicolon** (**`;`**), is called a **statement**. Most statements end with a semicolon. When the statement in line 9 executes, it displays `Welcome to Java Programming!` in the command window.

Using End-of-Line Comments on Right Braces for Readability

We include an end-of-line comment after a closing brace that ends a method declaration and after a closing brace that ends a class declaration. For example, line 10 indicates the closing brace of method `main`, and line 11 indicates the closing brace of class `Welcome1`.

Compiling and Executing Your First Java Application

We assume you're using the Java Development Kit's command-line tools, not an IDE. Our Java Resource Centers at `www.deitel.com/ResourceCenters.html` provide links to tutorials that help you get started with several popular Java development tools, including NetBeans™, Eclipse™ and others. We've also posted an Eclipse video at `www.deitel.com/books/AndroidHTP2/` to help you get started using this popular IDE.

To prepare to compile the program, open a command window and change to the directory where the program is stored. Many operating systems use the command `cd` to change directories. On Windows, for example,

```
cd c:\examples\appA\figA_01
```

changes to the `figA_01` directory. On UNIX/Linux/Max OS X, the command

```
cd ~/examples/appA/figA_01
```

changes to the `figA_01` directory.

To compile the program, type

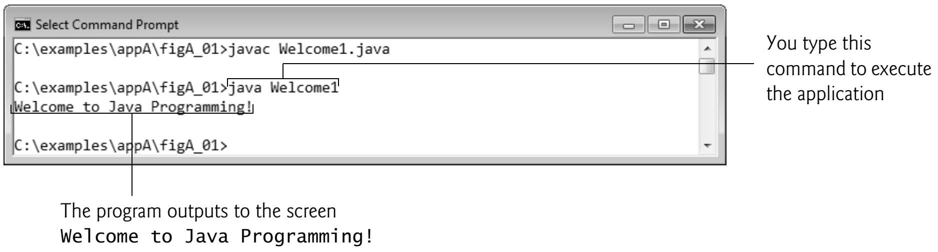
```
javac Welcome1.java
```

If the program contains no syntax errors, this command creates a new file called `Welcome1.class` (known as the **class file** for `Welcome1`) containing the platform-independent Java bytecodes that represent our application. When we use the `java` command to execute the application on a given platform, the JVM will translate these bytecodes into instructions that are understood by the underlying operating system and hardware.

Error-Prevention Tip A.1

When attempting to compile a program, if you receive a message such as "`bad command or filename`," "`javac: command not found`" or "`'javac' is not recognized as an internal or external command, operable program or batch file`," then your Java software installation was not completed properly. If you're using the JDK, this indicates that the system's `PATH` environment variable was not set properly. Please carefully review the installation instructions in the Before You Begin section of this book. On some systems, after correcting the `PATH`, you may need to reboot your computer or open a new command window for these settings to take effect.

Figure A.2 shows the program of Fig. A.1 executing in a Microsoft® Windows® 7 **Command Prompt** window. To execute the program, type `java Welcome1`. This command launches the JVM, which loads the `.class` file for class `Welcome1`. The command omits the `.class` file-name extension; otherwise, the JVM will not execute the program. The JVM calls method `main`. Next, the statement at line 9 of `main` displays `"Welcome to Java Programming!"`


Select Command Prompt
You type this command to execute the application
The program outputs to the screen Welcome to Java Programming!

```
C:\examples\appA\figA_01>javac Welcome1.java

C:\examples\appA\figA_01>java Welcome1
Welcome to Java Programming!

C:\examples\appA\figA_01>
```

Fig. A.2 | Executing `Welcome1` from the **Command Prompt**.

Error-Prevention Tip A.2

When attempting to run a Java program, if you receive a message such as "`Exception in thread "main" java.lang.NoClassDefFoundError: Welcome1`*," your* `CLASSPATH` *environment variable has not been set properly. Please carefully review the installation instructions in the Before You Begin section of this book. On some systems, you may need to reboot your computer or open a new command window after configuring the* `CLASSPATH`*.*

A.3 Modifying Your First Java Program

`Welcome to Java Programming!` can be displayed several ways. Class `Welcome2`, shown in Fig. A.3, uses two statements (lines 9–10) to produce the output shown in Fig. A.1.

```
 1  // Fig. A.3: Welcome2.java
 2  // Printing a line of text with multiple statements.
 3
 4  public class Welcome2
 5  {
 6     // main method begins execution of Java application
 7     public static void main( String[] args )
 8     {
 9        System.out.print( "Welcome to " );
10        System.out.println( "Java Programming!" );
11     } // end method main
12  } // end class Welcome2
```

```
Welcome to Java Programming!
```

Fig. A.3 | Printing a line of text with multiple statements.

The program is similar to Fig. A.1, so we discuss only the changes here. Line 2 is a comment stating the purpose of the program. Line 4 begins the `Welcome2` class declaration. Lines 9–10 of method `main` display one line of text. The first statement uses `System.out`'s method `print` to display a string. Each `print` or `println` statement resumes displaying characters from where the last `print` or `println` statement stopped displaying characters. Unlike `println`, after displaying its argument, `print` does *not* position the output cursor at the beginning of the next line in the command window—the next character the program displays will appear *immediately after* the last character that `print` displays. Thus, line 10 positions the first character in its argument (the letter "`J`") immediately after the last character that line 9 displays (the *space character* before the string's closing double-quote character).

Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using **newline characters**, which indicate to `System.out`'s `print` and `println` methods when to position the output cursor at the beginning of the next line in the command window. Like blank lines, space characters and tab characters, newline characters are white-space characters. The program in Fig. A.4 outputs four lines of text, using newline characters to determine when to begin each new line.

```
 1  // Fig. A.4: Welcome3.java
 2  // Printing multiple lines of text with a single statement.
 3
 4  public class Welcome3
 5  {
 6     // main method begins execution of Java application
 7     public static void main( String[] args )
 8     {
 9        System.out.println( "Welcome\nto\nJava\nProgramming!" );
10     } // end method main
11  } // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

Fig. A.4 | Printing multiple lines of text with a single statement.

Line 2 is a comment stating the program's purpose. Line 4 begins the `Welcome3` class declaration. Line 9 displays four separate lines of text in the command window. Normally, the characters in a string are displayed *exactly* as they appear in the double quotes. Note, however, that the paired characters \ and `n` (repeated three times in the statement) do not appear on the screen. The **backslash** (\) is an **escape character**. which has special meaning to `System.out`'s `print` and `println` methods. When a backslash appears in a string, Java combines it with the next character to form an **escape sequence**. The escape sequence `\n` represents the newline character. When a newline character appears in a string being output with `System.out`, the newline character causes the screen's output cursor to move to the beginning of the next line in the command window.

Figure A.5 lists several common escape sequences and describes how they affect the display of characters in the command window.

Escape sequence	Description
`\n`	Newline. Position the screen cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor at the beginning of the current line—do *not* advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
`\\`	Backslash. Used to print a backslash character.
`\"`	Double quote. Used to print a double-quote character. For example, `System.out.println( "\"in quotes\"" );` displays `"in quotes"`.

Fig. A.5 | Some common escape sequences.

A.4 Displaying Text with printf

The **System.out.printf** method displays formatted data. Figure A.6 uses this method to output the strings "Welcome to" and "Java Programming!". Lines 9–10 call method System.out.printf to display the program's output. The method call specifies three arguments—they're placed in a **comma-separated list**.

```
 1  // Fig. A.6: Welcome4.java
 2  // Displaying multiple lines with method System.out.printf.
 3
 4  public class Welcome4
 5  {
 6     // main method begins execution of Java application
 7     public static void main( String[] args )
 8     {
 9        System.out.printf( "%s\n%s\n",
10           "Welcome to", "Java Programming!" );
11     } // end method main
12  } // end class Welcome4
```

```
Welcome to
Java Programming!
```

Fig. A.6 | Displaying multiple lines with method System.out.printf.

Lines 9–10 represent only *one* statement. Java allows large statements to be split over many lines. We indent line 10 to indicate that it's a *continuation* of line 9.

Method printf's first argument is a **format string** that may consist of **fixed text** and **format specifiers**. Fixed text is output by printf just as it would be by print or println. Each format specifier is a placeholder for a value and specifies the type of data to output. Format specifiers also may include optional formatting information.

Format specifiers begin with a percent sign (%) followed by a character that represents the data type. For example, the format specifier **%s** is a placeholder for a string. The format string in line 9 specifies that printf should output two strings, each followed by a newline character. At the first format specifier's position, printf substitutes the value of the first argument after the format string. At each subsequent format specifier's position, printf substitutes the value of the next argument. So this example substitutes "Welcome to" for the first %s and "Java Programming!" for the second %s. The output shows that two lines of text are displayed.

A.5 Another Application: Adding Integers

Our next application reads (or inputs) two **integers** (whole numbers, such as –22, 7, 0 and 1024) typed by a user at the keyboard, computes their sum and displays it. Programs remember numbers and other data in the computer's memory and access that data through program elements called **variables**. The program of Fig. A.7 demonstrates these concepts. In the sample output, we use bold text to identify the user's input (i.e., **45** and **72**).

```
 1  // Fig. A.7: Addition.java
 2  // Addition program that displays the sum of two numbers.
 3  import java.util.Scanner; // program uses class Scanner
 4
 5  public class Addition
 6  {
 7     // main method begins execution of Java application
 8     public static void main( String[] args )
 9     {
10        // create a Scanner to obtain input from the command window
11        Scanner input = new Scanner( System.in );
12
13        int number1; // first number to add
14        int number2; // second number to add
15        int sum; // sum of number1 and number2
16
17        System.out.print( "Enter first integer: " ); // prompt
18        number1 = input.nextInt(); // read first number from user
19
20        System.out.print( "Enter second integer: " ); // prompt
21        number2 = input.nextInt(); // read second number from user
22
23        sum = number1 + number2; // add numbers, then store total in sum
24
25        System.out.printf( "Sum is %d\n", sum ); // display sum
26     } // end method main
27  } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. A.7 | Addition program that displays the sum of two numbers.

`Import` Declarations

Lines 1–2 state the figure number, file name and purpose of the program. A great strength of Java is its rich set of predefined classes that you can *reuse* rather than "reinventing the wheel." These classes are grouped into **packages**—named groups of related classes—and are collectively referred to as the **Java class library**, or the **Java Application Programming Interface** (**Java API**). Line 3 is an **`import` declaration** that helps the compiler locate a class that's used in this program. It indicates that this example uses Java's predefined `Scanner` class (discussed shortly) from package **`java.util`**.

Declaring Class `Addition`

Line 5 begins the declaration of class `Addition`. The file name for this `public` class must be `Addition.java`. Remember that the body of each class declaration starts with an opening left brace (line 6) and ends with a closing right brace (line 27).

The application begins execution with the `main` method (lines 8–26). The left brace (line 9) marks the beginning of method `main`'s body, and the corresponding right brace (line 26) marks its end. Method `main` is indented one level in the body of class `Addition`, and the code in the body of `main` is indented another level for readability.

Declaring and Creating a **`Scanner`** *to Obtain User Input from the Keyboard*

A **variable** is a location in the computer's memory where a value can be stored for use later in a program. All Java variables *must* be declared with a **name** and a **type** *before* they can be used. A variable's name enables the program to access the value of the variable in memory. A variable's name can be any valid identifier. A variable's type specifies what kind of information is stored at that location in memory. Like other statements, declaration statements end with a semicolon (`;`).

Line 11 is a **variable declaration statement** that specifies the name (`input`) and type (`Scanner`) of a variable that's used in this program. A **`Scanner`** enables a program to read data (e.g., numbers and strings) for use in a program. The data can come from many sources, such as the user at the keyboard or a file on disk. Before using a `Scanner`, you must create it and specify the source of the data.

The `=` in line 11 indicates that `Scanner` variable `input` should be **initialized** (i.e., prepared for use in the program) in its declaration with the result of the expression to the right of the equals sign—`new Scanner(System.in)`. This expression uses the **`new`** keyword to create a `Scanner` object that reads characters typed by the user at the keyboard. The **standard input object, `System.in`**, enables applications to read bytes of information typed by the user. The `Scanner` translates these bytes into types (like `int`s) that can be used in a program.

Declaring Variables to Store Integers

The variable declaration statements in lines 13–15 declare that variables `number1`, `number2` and `sum` hold data of type **`int`**—that is, integer values (whole numbers such as `72`, `-1127` and `0`). These variables are not yet initialized. The range of values for an `int` is –2,147,483,648 to +2,147,483,647. [*Note:* Actual `int` values may not contain commas.]

Other data types include **`float`** and **`double`**, for holding real numbers (such as `3.4`, `0.0` and `-11.19`), and **`char`**, for holding character data. Variables of type `char` represent individual characters, such as an uppercase letter (e.g., `A`), a digit (e.g., `7`), a special character (e.g., `*` or `%`) or an escape sequence (e.g., the newline character, `\n`). The types `int`, `float`, `double` and `char` are called **primitive types**. Primitive-type names are keywords and must appear in all lowercase letters. Appendix L summarizes the characteristics of the eight primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`).

Good Programming Practice A.2

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter.

Prompting the User for Input

Line 17 uses `System.out.print` to display the message `"Enter first integer: "`. This message is called a **prompt** because it directs the user to take a specific action. We use method `print` here rather than `println` so that the user's input appears on the same line as the prompt. Recall from Section A.2 that identifiers starting with capital letters typically represent class names. So, `System` is a class. Class `System` is part of package **`java.lang`**. Class `System` is not imported with an `import` declaration at the beginning of the program.

Software Engineering Observation A.1

By default, package `java.lang` is imported in every Java program; thus, classes in `java.lang` are the only ones in the Java API that do not require an `import` declaration.

Obtaining an `int` as Input from the User

Line 18 uses `Scanner` object `input`'s `nextInt` method to obtain an integer from the user at the keyboard. At this point the program waits for the user to type the number and press the *Enter* key to submit the number to the program.

Our program assumes that the user enters a valid integer value. If not, a runtime logic error will occur and the program will terminate. Appendix H discusses how to make your programs more robust by enabling them to handle such errors—this makes your program more *fault tolerant*.

In line 18, we place the result of the call to method `nextInt` (an `int` value) in variable `number1` by using the **assignment operator**, `=`. The statement is read as "`number1` gets the value of `input.nextInt()`." Operator `=` is called a **binary operator**, because it has two **operands**—`number1` and the result of the method call `input.nextInt()`. This statement is called an assignment statement, because it assigns a value to a variable. Everything to the *right* of the assignment operator, `=`, is always evaluated *before* the assignment is performed.

Good Programming Practice A.3

Placing spaces on either side of a binary operator makes the program more readable.

Prompting for and Inputting a Second `int`

Line 20 prompts the user to input the second integer. Line 21 reads the second integer and assigns it to variable `number2`.

Using Variables in a Calculation

Line 23 is an assignment statement that calculates the sum of the variables `number1` and `number2` then assigns the result to variable `sum` by using the assignment operator, `=`. The statement is read as "`sum` *gets* the value of `number1 + number2`." In general, calculations are performed in assignment statements. When the program encounters the addition operation, it performs the calculation using the values stored in the variables `number1` and `number2`. In the preceding statement, the addition operator is a *binary operator*—its *two* operands are the variables `number1` and `number2`. Portions of statements that contain calculations are called **expressions**. In fact, an expression is any portion of a statement that has a *value* associated with it. For example, the value of the expression `number1 + number2` is the *sum* of the numbers. Similarly, the value of the expression `input.nextInt()` is the integer typed by the user.

Displaying the Result of the Calculation

After the calculation has been performed, line 25 uses method `System.out.printf` to display the `sum`. The format specifier **`%d`** is a placeholder for an `int` value (in this case the value of `sum`)—the letter `d` stands for "decimal integer." The remaining characters in the format string are all fixed text. So, method `printf` displays `"Sum is "`, followed by the value of `sum` (in the position of the `%d` format specifier) and a newline.

Calculations can also be performed *inside* `printf` statements. We could have combined the statements at lines 23 and 25 into the statement

```
    System.out.printf( "Sum is %d\n", ( number1 + number2 ) );
```

The parentheses around the expression `number1 + number2` are not required—they're included to emphasize that the value of the *entire* expression is output in the position of the `%d` format specifier.

Java API Documentation

For each new Java API class we use, we indicate the package in which it's located. This information helps you locate descriptions of each package and class in the Java API documentation. A web-based version of this documentation can be found at

```
docs.oracle.com/javase/6/docs/api/
```

You can download it from

```
www.oracle.com/technetwork/java/javase/downloads/index.html
```

A.6 Memory Concepts

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

In the addition program of Fig. A.7, when the following statement (line 18) executes:

```
number1 = input.nextInt(); // read first number from user
```

the number typed by the user is placed into a memory location corresponding to the name `number1`. Suppose that the user enters `45`. The computer places that integer value into `number1` (Fig. A.8), replacing the previous value (if any) in that location. The previous value is lost.

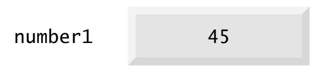


Fig. A.8 | Memory location showing the name and value of variable `number1`.

When the statement (line 21)

```
number2 = input.nextInt(); // read second number from user
```

executes, suppose that the user enters `72`. The computer places that integer value into location `number2`. The memory now appears as shown in Fig. A.9.

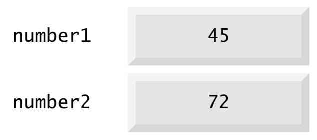


Fig. A.9 | Memory locations after storing values for `number1` and `number2`.

After the program of Fig. A.7 obtains values for `number1` and `number2`, it adds the values and places the total into variable `sum`. The statement (line 23)

```
sum = number1 + number2; // add numbers, then store total in sum
```

performs the addition, then replaces any previous value in `sum`. After `sum` has been calculated, memory appears as in Fig. A.10. `number1` and `number2` contain the values that were used in the calculation of sum. These values were used, but not destroyed, as the calculation was performed. When a value is read from a memory location, the process is nondestructive.

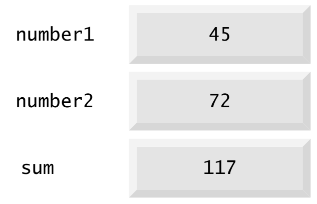


Fig. A.10 | Memory locations after storing the sum of number1 and number2.

A.7 Arithmetic

Most programs perform arithmetic calculations. The **arithmetic operators** are summarized in Fig. A.11. Note the use of various special symbols not used in algebra. The **asterisk** (*) indicates multiplication, and the percent sign (**%**) is the **remainder operator**, which we'll discuss shortly. The arithmetic operators in Fig. A.11 are *binary* operators, because each operates on *two* operands. For example, the expression f + 7 contains the binary operator + and the two operands f and 7.

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	`f + 7`
Subtraction	–	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Fig. A.11 | Arithmetic operators.

Integer division yields an integer quotient. For example, the expression 7 / 4 evaluates to 1, and the expression 17 / 5 evaluates to 3. Any fractional part in integer division is simply *discarded* (i.e., *truncated*)—no rounding occurs. Java provides the remainder operator, %, which yields the remainder after division. The expression x % y yields the remainder after x is divided by y. Thus, 7 % 4 yields 3, and 17 % 5 yields 2. This operator is most commonly used with integer operands but can also be used with other arithmetic types.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in Java must be written in **straight-line form** to facilitate entering programs into the computer. Thus, expressions such as "a divided by b" must be written as `a / b`, so that all constants, variables and operators appear in a straight line. The following algebraic notation is generally not acceptable to compilers:

$$\frac{a}{b}$$

Parentheses for Grouping Subexpressions

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

```
      a * ( b + c )
```

If an expression contains **nested parentheses**, such as

```
      ( ( a + b ) * c )
```

the expression in the innermost set of parentheses (`a + b` in this case) is evaluated first.

Rules of Operator Precedence

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

1. Multiplication, division and remainder operations are applied first. If an expression contains several such operations, they're applied from left to right. Multiplication, division and remainder operators have the same level of precedence.
2. Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.

These rules enable Java to apply operators in the correct order.[1] When we say that operators are applied from left to right, we're referring to their **associativity**. Some operators associate from right to left. Figure A.12 summarizes these rules of operator precedence. A complete precedence chart is included in Appendix K.

Operator(s)	Operation(s)	Order of evaluation (precedence)

`*`
`/`
`%`
	Multiplication
	Division
	Remainder
		Evaluated first. If there are several operators of this type, they're evaluated from left to right.

`+` `-`	Addition Subtraction	Evaluated next. If there are several operators of this type, they're evaluated from left to right.
`=`	Assignment	Evaluated last.

Fig. A.12 | Precedence of arithmetic operators.

1. We use simple examples to explain the order of evaluation of expressions. Subtle issues occur in the more complex expressions you'll encounter. For more information on order of evaluation, see Chapter 15 of *The Java™ Language Specification* (`java.sun.com/docs/books/jls/`).

Sample Algebraic and Java Expressions

Now let's consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its Java equivalent. The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \dfrac{a + b + c + d + e}{5}$

Java:
```
m = ( a + b + c + d + e ) / 5;
```

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

$$a + b + c + d + \frac{e}{5}$$

Here's an example of the equation of a straight line:

Algebra: $y = mx + b$

Java:
```
y = m * x + b;
```

No parentheses are required. The multiplication operator is applied first because multiplication has a higher precedence than addition. The assignment occurs last because it has a lower precedence than multiplication or addition.

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

Algebra: $z = pr\%q + w/x - y$

Java:
```
z = p * r % q + w / x - y;
  6   1   2   4   3   5
```

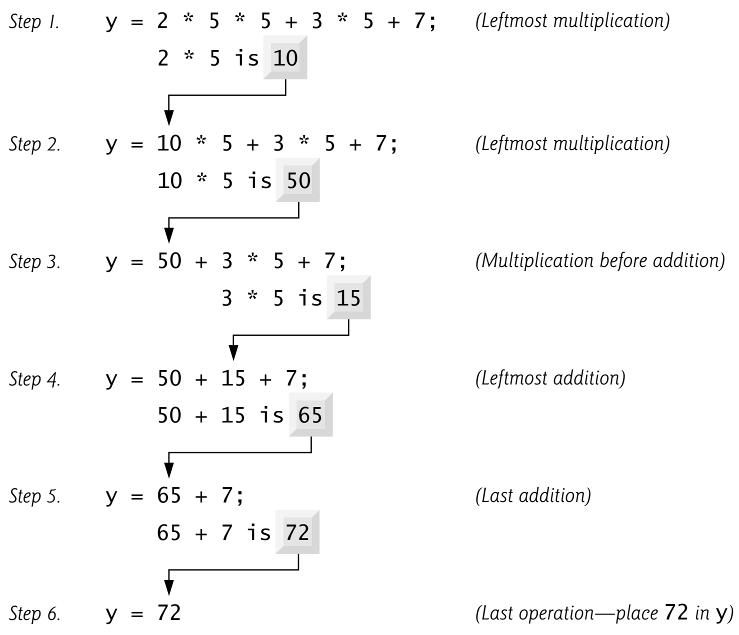
The circled numbers under the statement indicate the order in which Java applies the operators. The *, % and / operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than + and -. The + and - operations are evaluated next. These operations are also applied from left to right. The assignment (=) operation is evaluated last.

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, consider the evaluation of an assignment expression that includes a second-degree polynomial $ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
  6   1   2   4   3   5
```

The multiplication operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than addition. (Java has no arithmetic operator for exponentiation in Java, so x^2 is represented as x * x. Section C.16 shows an alternative for performing exponentiation.) The addition operations are evaluated next from left to right. Suppose that a, b, c and x are initialized (given values) as follows: a = 2, b = 3, c = 7 and x = 5. Figure A.13 illustrates the order in which the operators are applied.



```
Step 1.   y = 2 * 5 * 5 + 3 * 5 + 7;      (Leftmost multiplication)
              2 * 5 is 10

Step 2.   y = 10 * 5 + 3 * 5 + 7;         (Leftmost multiplication)
              10 * 5 is 50

Step 3.   y = 50 + 3 * 5 + 7;             (Multiplication before addition)
                   3 * 5 is 15

Step 4.   y = 50 + 15 + 7;                (Leftmost addition)
              50 + 15 is 65

Step 5.   y = 65 + 7;                     (Last addition)
              65 + 7 is 72

Step 6.   y = 72                          (Last operation—place 72 in y)
```

Fig. A.13 | Order in which a second-degree polynomial is evaluated.

A.8 Decision Making: Equality and Relational Operators

A **condition** is an expression that can be **true** or **false**. This section introduces Java's **if selection statement**, which allows a program to make a **decision** based on a condition's value. For example, the condition "grade is greater than or equal to 60" determines whether a student passed a test. If the condition in an if statement is true, the body of the if statement executes. If the condition is false, the body does not execute. We'll see an example shortly.

Conditions in if statements can be formed by using the **equality operators** (== and !=) and **relational operators** (>, <, >= and <=) summarized in Fig. A.14. Both equality operators have the same level of precedence, which is *lower* than that of the relational operators. The equality operators associate from left to right. The relational operators all have the same level of precedence and also associate from left to right.

Standard algebraic equality or relational operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

Fig. A.14 | Equality and relational operators. (Part 1 of 2.)

Standard algebraic equality or relational operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Relational operators			
>	`>`	`x > y`	`x` is greater than `y`
<	`<`	`x < y`	`x` is less than `y`
≥	`>=`	`x >= y`	`x` is greater than or equal to `y`
≤	`<=`	`x <= y`	`x` is less than or equal to `y`

Fig. A.14 | Equality and relational operators. (Part 2 of 2.)

Figure A.15 uses six `if` statements to compare two integers input by the user. If the condition in any of these `if` statements is true, the statement associated with that `if` statement executes; otherwise, the statement is skipped. We use a `Scanner` to input the integers from the user and store them in variables `number1` and `number2`. The program compares the numbers and displays the results of the comparisons that are true.

```java
// Fig. A.15: Comparison.java
// Compare integers using if statements, relational operators
// and equality operators.
import java.util.Scanner; // program uses class Scanner

public class Comparison
{
   // main method begins execution of Java application
   public static void main( String[] args )
   {
      // create Scanner to obtain input from command line
      Scanner input = new Scanner( System.in );

      int number1; // first number to compare
      int number2; // second number to compare

      System.out.print( "Enter first integer: " ); // prompt
      number1 = input.nextInt(); // read first number from user

      System.out.print( "Enter second integer: " ); // prompt
      number2 = input.nextInt(); // read second number from user

      if ( number1 == number2 )
         System.out.printf( "%d == %d\n", number1, number2 );

      if ( number1 != number2 )
         System.out.printf( "%d != %d\n", number1, number2 );

```

Fig. A.15 | Compare integers using `if` statements, relational operators and equality operators. (Part 1 of 2.)

```
29          if ( number1 < number2 )
30             System.out.printf( "%d < %d\n", number1, number2 );
31
32          if ( number1 > number2 )
33             System.out.printf( "%d > %d\n", number1, number2 );
34
35          if ( number1 <= number2 )
36             System.out.printf( "%d <= %d\n", number1, number2 );
37
38          if ( number1 >= number2 )
39             System.out.printf( "%d >= %d\n", number1, number2 );
40       } // end method main
41    } // end class Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. A.15 | Compare integers using `if` statements, relational operators and equality operators. (Part 2 of 2.)

The declaration of class `Comparison` begins at line 6. The class's `main` method (lines 9–40) begins the execution of the program. Line 12 declares `Scanner` variable `input` and assigns it a `Scanner` that inputs data from the standard input (i.e., the keyboard).

Lines 14–15 declare the `int` variables used to store the values input from the user.

Lines 17–18 prompt the user to enter the first integer and input the value, respectively. The input value is stored in variable `number1`.

Lines 20–21 prompt the user to enter the second integer and input the value, respectively. The input value is stored in variable `number2`.

Lines 23–24 compare the values of `number1` and `number2` to determine whether they're equal. An `if` statement always begins with keyword `if`, followed by a condition in parentheses. An `if` statement expects one statement in its body, but may contain multiple statements if they're enclosed in a set of braces (`{}`). The indentation of the body statement shown here is not required, but it improves the program's readability by emphasizing that the statement in line 24 *is part of* the `if` statement that begins at line 23. Line 24 executes only if the numbers stored in variables `number1` and `number2` are equal (i.e., the condition is true). The `if` statements in lines 26–27, 29–30, 32–33, 35–36 and 38–39 compare `number1` and `number2` using the operators `!=`, `<`, `>`, `<=` and `>=`, respectively. If the condition in one or more of the `if` statements is true, the corresponding body statement executes.

Common Programming Error A.3

Confusing the equality operator, ==, with the assignment operator, =, can cause a logic error or a syntax error. The equality operator should be read as "is equal to" and the assignment operator as "gets" or "gets the value of." To avoid confusion, some people read the equality operator as "double equals" or "equals equals."

There's no semicolon (`;`) at the end of the first line of each `if` statement. Such a semicolon would result in a logic error at execution time. For example,

```
if ( number1 == number2 ); // logic error
   System.out.printf( "%d == %d\n", number1, number2 );
```

would actually be interpreted by Java as

```
if ( number1 == number2 )
   ; // empty statement

System.out.printf( "%d == %d\n", number1, number2 );
```

where the semicolon on the line by itself—called the **empty statement**—is the statement to execute if the condition in the `if` statement is true. When the empty statement executes, no task is performed. The program then continues with the output statement, which always executes, regardless of whether the condition is true or false, because the output statement is not part of the `if` statement.

Note the use of white space in Fig. A.15. Recall that the compiler normally ignores white space. So, statements may be split over several lines and may be spaced according to your preferences without affecting a program's meaning. It's incorrect to split identifiers and strings. Ideally, statements should be kept small, but this is not always possible.

Figure A.16 shows the operators discussed so far in decreasing order of precedence. All but the assignment operator, `=`, associate from left to right. The assignment operator, `=`, associates from right to left, so an expression like `x = y = 0` is evaluated as if it had been written as `x = (y = 0)`, which first assigns the value `0` to variable `y`, then assigns the result of that assignment, `0`, to `x`.

Operators	Associativity	Type
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`=`	right to left	assignment

Fig. A.16 | Precedence and associativity of operators discussed.

A.9 Wrap-Up

In this appendix, you learned many important features of Java, including displaying data on the screen in a **Command Prompt**, inputting data from the keyboard, performing calculations and making decisions. The applications presented here introduced you to basic programming concepts. As you'll see in Appendix B, Java applications typically contain just a few lines of code in method `main`—these statements normally create the objects that perform the work of the application. In Appendix B, you'll learn how to implement your own classes and use objects of those classes in applications.

Self-Review Exercises

A.1 Fill in the blanks in each of the following statements:

a) A(n) ________ begins the body of every method, and a(n) ________ ends the body of every method.
b) The ________ statement is used to make decisions.
c) ________ begins an end-of-line comment.
d) The ________ object is known as the standard output object.
e) ________ are reserved for use by Java.
f) Java applications begin execution at method ________.
g) The ________ after the identifier `main` indicate that it's a program building block called a method.

A.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Comments cause the computer to print the text after the `//` on the screen when the program executes.
b) All variables must be given a type when they're declared.
c) Java considers the variables `number` and `NuMbEr` to be identical.
d) The remainder operator (`%`) can be used only with integer operands.
e) The arithmetic operators `*`, `/`, `%`, `+` and `-` all have the same level of precedence.

A.3 Write statements to accomplish each of the following tasks:

a) Declare variables `c`, `thisIsAVariable`, `q76354` and `number` to be of type `int`.
b) Prompt the user to enter an integer.
c) Input an integer and assign the result to `int` variable `value`. Assume `Scanner` variable `input` can be used to read a value from the keyboard.
d) Print `"This is a Java program"` on one line in the command window. Use method `System.out.println`.
e) Print `"This is a Java program"` on two lines in the command window. The first line should end with `Java`. Use method `System.out.println`.
f) Print `"This is a Java program"` on two lines in the command window. The first line should end with `Java`. Use method `System.out.printf` and two `%s` format specifiers.
g) If the variable `number` is not equal to `7`, display `"The variable number is not equal to 7"`.

A.4 Identify and correct the errors in each of the following statements:

a)
```
if ( c >= 10 );
   System.out.println( "c is greater than or equal to 10" );
```
b)
```
if ( c =! 7 )
   System.out.println( "c is not equal to 7" );
```

A.5 Write declarations, statements or comments that accomplish each of the following tasks:

a) State that a program will calculate the average of three numbers.

b) Create a `Scanner` called `input` that reads values from the standard input.
c) Declare the variables `x`, `y`, `z` and `result` to be of type int.
d) Prompt the user to enter the first number.
e) Read the first number from the user and store it in the variable `x`.
f) Prompt the user to enter the second number.
g) Read the second number from the user and store it in the variable `y`.
h) Prompt the user to enter the third number.
i) Read the third number from the user and store it in the variable `z`.
j) Compute the average of the three numbers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.
k) Display the message "`Average is`" followed by the value of the variable `result`.

A.6 Using the statements you wrote in Exercise A.5, write a complete program that calculates and prints the average of three numbers.

Answers to Self-Review Exercises

A.1 a) left brace (`{`), right brace (`}`). b) `if`. c) `//`. d) **`System.out.`** e) Keywords. f) `main`. g) **parenthesis.**

A.2 a) False. Comments do not cause any action to be performed when the program executes. They're used to document programs and improve their readability. b) True. c) False. Java is case sensitive, so these variables are distinct. d) False. The remainder operator can also be used with non-integer operands in Java. e) False. The operators `*`, `/` and `%` are higher precedence than operators `+` and `-`.

A.3 a) `int c, thisIsAVariable, q76354, number;`
or
```
int c;
int thisIsAVariable;
int q76354;
int number;
```
b) `System.out.print( "Enter an integer: " );`
c) `value = input.nextInt();`
d) `System.out.println( "This is a Java program" );`
e) `System.out.println( "This is a Java\nprogram" );`
f) `System.out.printf( "%s\n%s\n", "This is a Java", "program" );`
g)
```
if ( number != 7 )
   System.out.println( "The variable number is not equal to 7" );
```

A.4 a) Error: Semicolon after the right parenthesis of the condition `( c >= 10 )` in the `if`. Correction: Remove the semicolon after the right parenthesis. [*Note*: As a result, the output statement will execute regardless of whether the condition in the `if` is true.]
b) Error: The relational operator =! is incorrect.
Correction: Change =! to !=.

A.5 a) `// Calculate the average of three numbers`
b) `Scanner input = new Scanner( System.in );`
c) `float x, y, z, result;`
or
```
float x;
float y;
float z;
float result;
```

d) `System.out.print( "Enter first number: "  );`
e) `x = input.nextFloat();`
f) `System.out.print( "Enter second number: " );`
g) `y = input.nextFloat();`
h) `System.out.print( "Enter third number: " );`
i) `z = input.nextFloat();`
j) `result = (x * y * z)/3;`
k) `System.out.printf( "Average is %f\n", result );`

A.6 The solution to Self-Review Exercise A.6 is as follows:

```java
// Ex. 2.6: Average.java
// Calculate the average of three numbers.
import java.util.Scanner; // program uses Scanner

public class Average
{
   public static void main( String[] args )
   {
       // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      float x; // first number input by user
      float y; // second number input by user
      float z; // third number input by user
      float result; // average of numbers

      System.out.print( "Enter first number: " ); // prompt for input
      x = input.nextFloat(); // read first number

      System.out.print( "Enter second number: " ); // prompt for input
      y = input.nextFloat(); // read second number

      System.out.print( "Enter third number: " ); // prompt for input
      z = input.nextFloat(); // read third number

      result = (x + y + z)/3; // calculate average of numbers

      System.out.printf( "Average is %f\n", result );
   } // end method main
} // end class Average
```

```
Enter first number: 15.5
Enter second number: 25.5
Enter third number: 20.0
Average is 30.5
```

Exercises

A.7 Fill in the blanks in each of the following statements:

a) In Java, blank lines, space characters and tabs together are known as ________.
b) A decision can be made in a Java program with a(n) ________.
c) The ________ is an escape character, which has special meaning to `print` and `println` methods.
d) The arithmetic operators with the same precedence as multiplication are ________ and ________.
e) When parentheses in an arithmetic expression are nested, the ________ set of parentheses is evaluated first.

f) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a(n) ________.

A.8 Write Java statements that accomplish each of the following tasks:

a) Display the message `"Enter a real number: "`, leaving the cursor on the next line.
b) Assign the sum of variables `a` and `b` to variable `c`.
c) Use conditional operator to find maximum value of any 3 integers.

A.9 State whether each of the following is *true* or *false*. If *false*, explain why.

a) In Java, `\r` is used as new line escape sequence.
b) In an arithmetic expression, `Addition` and `Subtraction` operations are evaluated first. If there are several operators of this type, they're evaluated from left to right.
c) A valid Java arithmetic expression with no parentheses is evaluated from left to right.
d) The following are all invalid variable names: `3g`, `87`, `67h2`, `h22` and `2h`.

A.10 Assuming that `a = 5` and `b = 6`, what does each of the following statements display?

a) `System.out.printf( "a = %f\n", a );`
b) `System.out.printf( "Product of %d and %d is %d\n", a, b, ( a * b ) );`
c) `System.out.printf( "Among %d & %d, %d is greater", a, b,( a > b ? a : b );`
d) `System.out.printf(  "%d = %d \n", ( a * b ), ( b * a ) );`

A.11 *(Arithmetic, Smallest and Largest)* Write an application that inputs three integers from the user and displays the sum, average, product, smallest and largest of the numbers. Use the techniques shown in Fig. A.15. [*Note:* The calculation of the average in this exercise should result in an integer representation of the average. So, if the sum of the values is 7, the average should be 2, not 2.3333….]

A.12 What does the following code print?

```
System.out.printf( "%s\n%s\n%s\n", "*", "***", "*****" );
```

A.13 *(Largest and Smallest Integers)* Write an application that reads five integers and determines and prints the largest and smallest integers in the group. Use only the programming techniques you learned in this appendix.

A.14 *(Odd or Even)* Write an application that reads an integer and determines and prints whether it's odd or even. [*Hint:* Use the remainder operator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of 0 when divided by 2.]

A.15 *(Multiples)* Write an application that reads two integers, determines whether the first is a multiple of the second and prints the result. [*Hint:* Use the remainder operator.]

A.16 *(Diameter, Circumference and Area of a Circle)* Here's a peek ahead. In this appendix, you learned about integers and the type `int`. Java can also represent floating-point numbers that contain decimal points, such as 3.14159. Write an application that inputs from the user the radius of a circle as an integer and prints the circle's diameter, circumference and area using the floating-point value 3.14159 for π. Use the techniques shown in Fig. A.7. [*Note:* You may also use the predefined constant `Math.PI` for the value of π. This constant is more precise than the value 3.14159. Class `Math` is defined in package `java.lang`. Classes in that package are imported automatically, so you do not need to `import` class `Math` to use it.] Use the following formulas (r is the radius):

$$\textit{diameter} = 2r$$
$$\textit{circumference} = 2\pi r$$
$$\textit{area} = \pi r^2$$

Do not store the results of each calculation in a variable. Rather, specify each calculation as the value that will be output in a `System.out.printf` statement. The values produced by the circumference and area calculations are floating-point numbers. Such values can be output with the format specifier `%f` in a `System.out.printf` statement. You'll learn more about floating-point numbers in Appendix B.

A.17 *(Separating the Digits in an Integer)* Write an application that inputs one number consisting of five digits from the user, separates the number into its individual digits and prints the digits separated from one another by three spaces each. For example, if the user types in the number `42339`, the program should print

```
4   2   3   3   9
```

Assume that the user enters the correct number of digits. What happens when you execute the program and type a number with more than five digits? What happens when you execute the program and type a number with fewer than five digits? [*Hint:* It's possible to do this exercise with the techniques you learned in this appendix. You'll need to use both division and remainder operations to "pick off" each digit.]

A.18 *(Table of Squares and Cubes)* Using only the programming techniques you learned in this appendix, write an application that calculates the squares and cubes of the numbers from 0 to 10 and prints the resulting values in table format, as shown below. [*Note:* This program does not require any input from the user.]

```
number  square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

B Introduction to Classes, Objects, Methods and Strings

Objectives

In this appendix you'll learn:

- How to declare a class and use it to create an object.
- How to implement a class's behaviors as methods.
- How to implement a class's attributes as instance variables and properties.
- How to call an object's methods to make them perform their tasks.
- What instance variables of a class and local variables of a method are.
- How to use a constructor to initialize an object's data.
- The differences between primitive and reference types.

Outline

B.1 Introduction

In this appendix, we introduce some key concepts of object-oriented programming in Java, including classes, objects, methods, instance variables and constructors. We explore the differences between primitive types and reference types, and we present a simple framework for organizing object-oriented applications.

B.2 Declaring a Class with a Method and Instantiating an Object of a Class

In this section, you'll create a *new* class, then use it to create an object. We begin by declaring classes `GradeBook` (Fig. B.1) and `GradeBookTest` (Fig. B.2). Class `GradeBook` (declared in the file `GradeBook.java`) will be used to display a message on the screen (Fig. B.2) welcoming the instructor to the grade book application. Class `GradeBookTest` (declared in the file `GradeBookTest.java`) is an application class in which the `main` method will create and use an object of class `GradeBook`. *Each class declaration that begins with keyword* `public` *must be stored in a file having the same name as the class and ending with the* `.java` *file-name extension.* Thus, classes `GradeBook` and `GradeBookTest` must be declared in *separate* files, because each class is declared `public`.

Class GradeBook

The `GradeBook` class declaration (Fig. B.1) contains a `displayMessage` method (lines 7–10) that displays a message on the screen. We'll need to make an object of this class and call its method to execute line 9 and display the message.

```
 1  // Fig. B.1: GradeBook.java
 2  // Class declaration with one method.
 3
 4  public class GradeBook
 5  {
 6     // display a welcome message to the GradeBook user
 7     public void displayMessage()
 8     {
 9        System.out.println( "Welcome to the Grade Book!" );
10     } // end method displayMessage
11  } // end class GradeBook
```

Fig. B.1 | Class declaration with one method.

The *class declaration* begins in line 4. The keyword `public` is an **access modifier**. For now, we'll simply declare every class `public`. Every class declaration contains keyword `class` followed immediately by the class's name. Every class's body is enclosed in a pair of left and right braces, as in lines 5 and 11 of class `GradeBook`.

In Appendix A, each class we declared had one method named `main`. Class `GradeBook` also has one method—`displayMessage` (lines 7–10). Recall that `main` is a special method that's *always* called automatically by the Java Virtual Machine (JVM) when you execute an application. Most methods do not get called automatically. As you'll soon see, you must call method `displayMessage` explicitly to tell it to perform its task.

The method declaration begins with keyword `public` to indicate that the method is "available to the public"—it can be called from methods of other classes. Next is the method's **return type**, which specifies the type of data the method returns to its caller after performing its task. The return type `void` indicates that this method will perform a task but will *not* return (i.e., give back) any information to its **calling method**. You've used methods that return information—for example, in Appendix A you used `Scanner` method `nextInt` to input an integer typed by the user at the keyboard. When `nextInt` reads a value from the user, it returns that value for use in the program.

The name of the method, `displayMessage`, follows the return type. By convention, method names begin with a lowercase first letter and subsequent words in the name begin with a capital letter. The *parentheses* after the method name indicate that this is a *method*. Empty parentheses, as in line 7, indicate that this method does not require additional information to perform its task. Line 7 is commonly referred to as the **method header**. Every method's body is delimited by left and right braces, as in lines 8 and 10.

The body of a method contains one or more statements that perform the method's task. In this case, the method contains one statement (line 9) that displays the message `"Welcome to the Grade Book!"` followed by a newline (because of `println`) in the command window. After this statement executes, the method has completed its task.

Class `GradeBookTest`

Next, we'll use class `GradeBook` in an application. As you learned in Appendix A, method `main` begins the execution of *every* application. A class that contains method `main` begins the execution of a Java application. Class `GradeBook` is *not* an application because it does *not* contain `main`. Therefore, if you try to execute `GradeBook` by typing `java GradeBook` in the command window, an error will occur. To fix this problem, we must either declare a separate class that contains a `main` method or place a `main` method in class `GradeBook`. To help you prepare for the larger programs you'll encounter later in this book and in industry, we use a separate class (`GradeBookTest` in this example) containing method `main` to test each new class we create. Some programmers refer to such a class as a *driver class*. The `GradeBookTest` class declaration (Fig. B.2) contains the `main` method that will control our application's execution.

Lines 7–14 declare method `main`. A key part of enabling the JVM to locate and call method `main` to begin the application's execution is the `static` keyword (line 7), which indicates that `main` is a `static` method. *A `static` method is special, because you can call it without first creating an object of the class in which the method is declared.* We discuss `static` methods in Appendix D.

```java
 1  // Fig. B.2: GradeBookTest.java
 2  // Creating a GradeBook object and calling its displayMessage method.
 3
 4  public class GradeBookTest
 5  {
 6     // main method begins program execution
 7     public static void main( String[] args )
 8     {
 9        // create a GradeBook object and assign it to myGradeBook
10        GradeBook myGradeBook = new GradeBook();
11
12        // call myGradeBook's displayMessage method
13        myGradeBook.displayMessage();
14     } // end main
15  } // end class GradeBookTest
```

```
Welcome to the Grade Book!
```

Fig. B.2 | Creating a `GradeBook` object and calling its `displayMessage` method.

In this application, we'd like to call class `GradeBook`'s `displayMessage` method to display the welcome message in the command window. Typically, you cannot call a method that belongs to another class until you create an object of that class, as shown in line 10. We begin by declaring variable `myGradeBook`. The variable's type is `GradeBook`—the class we declared in Fig. B.1. Each new *class* you create becomes a new *type* that can be used to declare variables and create objects.

Variable `myGradeBook` is initialized (line 10) with the result of the **class instance creation expression** `new GradeBook()`. Keyword **`new`** creates a new object of the class specified to the right of the keyword (i.e., `GradeBook`). The parentheses to the right of `GradeBook` are required. As you'll learn in Section B.6, those parentheses in combination with a class name represent a call to a **constructor**, which is similar to a method but is used only at the time an object is *created* to *initialize* the object's data. You'll see that data can be placed in the parentheses to specify *initial values* for the object's data. For now, we simply leave the parentheses empty.

Just as we can use object `System.out` to call its methods `print`, `printf` and `println`, we can use object `myGradeBook` to call its method `displayMessage`. Line 13 calls the method `displayMessage` (lines 7–10 of Fig. B.1) using `myGradeBook` followed by a **dot separator** (`.`), the method name `displayMessage` and an empty set of parentheses. This call causes the `displayMessage` method to perform its task. This method call differs from those in Appendix A that displayed information in a command window—each of those method calls provided arguments that specified the data to display. At the beginning of line 13, "`myGradeBook.`" indicates that `main` should use the `myGradeBook` object that was created in line 10. Line 7 of Fig. B.1 indicates that method `displayMessage` has an *empty parameter list*—that is, `displayMessage` does *not* require additional information to perform its task. For this reason, the method call (line 13 of Fig. B.2) specifies an empty set of parentheses after the method name to indicate that *no arguments* are being passed to method `displayMessage`. When method `displayMessage` completes its task, method `main` continues executing at line 14. This is the end of method `main`, so the program terminates.

Any class can contain a main method. The JVM invokes the main method *only* in the class used to execute the application. If an application has multiple classes that contain main, the one that's invoked is the one in the class named in the java command.

Compiling an Application with Multiple Classes

You must compile the classes in Fig. B.1 and Fig. B.2 before you can execute the application. First, change to the directory that contains the application's source-code files. Next, type the command

```
javac GradeBook.java GradeBookTest.java
```

to compile *both* classes at once. If the directory containing the application includes only this application's files, you can compile *all* the classes in the directory with the command

```
javac *.java
```

The asterisk (*) in *.java indicates that *all* files in the current directory that end with the file-name extension ".java" should be compiled.

B.3 Declaring a Method with a Parameter

In our car analogy from Section 1.8, we discussed the fact that pressing a car's gas pedal sends a *message* to the car to *perform a task*—to go faster. But *how fast* should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates. So the message to the car actually includes the *task to perform* and *additional information* that helps the car perform the task. This additional information is known as a **parameter**—the value of the parameter helps the car determine how fast to accelerate. Similarly, a method can require one or more parameters that represent additional information it needs to perform its task. Parameters are defined in a comma-separated **parameter list**, which is located inside the parentheses that follow the method name. Each parameter must specify a *type* and a variable name. The parameter list may contain any number of parameters, including none at all. Empty parentheses following the method name (as in Fig. B.1, line 7) indicate that a method does *not* require any parameters.

Arguments to a Method

A method call supplies values—called *arguments*—for each of the method's parameters. For example, the method System.out.println requires an argument that specifies the data to output in a command window. Similarly, to make a deposit into a bank account, a deposit method specifies a parameter that represents the deposit amount. When the deposit method is called, an argument value representing the deposit amount is assigned to the method's parameter. The method then makes a deposit of that amount.

Class Declaration with a Method That Has One Parameter

We now declare class GradeBook (Fig. B.3) with a displayMessage method that displays the course name as part of the welcome message. (See the sample execution in Fig. B.4.) The new method requires a parameter that represents the course name to output.

Before discussing the new features of class GradeBook, let's see how the new class is used from the main method of class GradeBookTest (Fig. B.4). Line 12 creates a Scanner named input for reading the course name from the user. Line 15 creates the GradeBook object `myGradeBook`. Line 18 prompts the user to enter a course name. Line 19 reads the name from the user and assigns it to the `nameOfCourse` variable, using `Scanner` method **`nextLine`** to perform the input. The user types the course name and presses *Enter* to submit the course name to the program. Pressing *Enter* inserts a newline character at the end of the characters typed by the user. Method `nextLine` reads characters typed by the user until it encounters the newline character, then returns a `String` containing the characters up to, but *not* including, the newline. The newline character is *discarded*.

```
 1  // Fig. B.3: GradeBook.java
 2  // Class declaration with one method that has a parameter.
 3
 4  public class GradeBook
 5  {
 6     // display a welcome message to the GradeBook user
 7     public void displayMessage( String courseName )
 8     {
 9        System.out.printf( "Welcome to the grade book for\n%s!\n",
10           courseName );
11     } // end method displayMessage
12  } // end class GradeBook
```

Fig. B.3 | Class declaration with one method that has a parameter.

```
 1  // Fig. B.4: GradeBookTest.java
 2  // Create a GradeBook object and pass a String to
 3  // its displayMessage method.
 4  import java.util.Scanner; // program uses Scanner
 5
 6  public class GradeBookTest
 7  {
 8     // main method begins program execution
 9     public static void main( String[] args )
10     {
11        // create Scanner to obtain input from command window
12        Scanner input = new Scanner( System.in );
13
14        // create a GradeBook object and assign it to myGradeBook
15        GradeBook myGradeBook = new GradeBook();
16
17        // prompt for and input course name
18        System.out.println( "Please enter the course name:" );
19        String nameOfCourse = input.nextLine(); // read a line of text
20        System.out.println(); // outputs a blank line
21
22        // call myGradeBook's displayMessage method
23        // and pass nameOfCourse as an argument
24        myGradeBook.displayMessage( nameOfCourse );
25     } // end main
26  } // end class GradeBookTest
```

Fig. B.4 | Create a `GradeBook` object and pass a `String` to its `displayMessage` method. (Part 1 of 2.)

```
Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. B.4 | Create a `GradeBook` object and pass a `String` to its `displayMessage` method. (Part 2 of 2.)

Class `Scanner` also provides method **`next`** that reads individual words. When the user presses *Enter* after typing input, method `next` reads characters until it encounters a *white-space character* (such as a space, tab or newline), then returns a `String` containing the characters up to, but *not* including, the white-space character (which is discarded). All information after the first white-space character is not lost—it can be read by other statements that call the `Scanner`'s methods later in the program. Line 20 outputs a blank line.

Line 24 calls `myGradeBooks`'s `displayMessage` method. The variable `nameOfCourse` in parentheses is the *argument* that's passed to method `displayMessage` so that the method can perform its task. The value of variable `nameOfCourse` in `main` becomes the value of method `displayMessage`'s *parameter* `courseName` in line 7 of Fig. B.3. When you execute this application, notice that method `displayMessage` outputs the name you type as part of the welcome message (Fig. B.4).

More on Arguments and Parameters

In Fig. B.3, `displayMessage`'s parameter list (line 7) declares one parameter indicating that the method requires a `String` to perform its task. When the method is called, the argument value in the call is assigned to the corresponding parameter (`courseName`) in the method header. Then, the method body uses the value of the `courseName` parameter. Lines 9–10 of Fig. B.3 display parameter `courseName`'s value, using the `%s` format specifier in `printf`'s format string. The parameter variable's name (`courseName` in Fig. B.3, line 7) can be the *same or different* from the argument variable's name (`nameOfCourse` in Fig. B.4, line 24).

The number of arguments in a method call *must* match the number of parameters in the parameter list of the method's declaration. Also, the argument types in the method call must be "consistent with" the types of the corresponding parameters in the method's declaration. (As you'll learn in Appendix D, an argument's type and its corresponding parameter's type are not always required to be *identical*.) In our example, the method call passes one argument of type `String` (`nameOfCourse` is declared as a `String` in line 19 of Fig. B.4) and the method declaration specifies one parameter of type `String` (`courseName` is declared as a `String` in line 7 of Fig. B.3). So in this example the type of the argument in the method call exactly matches the type of the parameter in the method header.

*Notes on **`import`** Declarations*

Notice the `import` declaration in Fig. B.4 (line 4). This indicates to the compiler that the program uses class `Scanner`. Why do we need to import class `Scanner`, but not classes `System`, `String` or `GradeBook`? Classes `System` and `String` are in package `java.lang`, which is implicitly imported into *every* Java program, so all programs can use that package's classes *without* explicitly importing them. Most other classes you'll use in Java programs must be imported explicitly.

There's a special relationship between classes that are compiled in the same directory on disk, like classes `GradeBook` and `GradeBookTest`. By default, such classes are considered to be in the same package—known as the **default package**. Classes in the same package are *implicitly imported* into the source-code files of other classes in the same package. Thus, an `import` declaration is *not* required when one class in a package uses another in the same package—such as when class `GradeBookTest` uses class `GradeBook`.

The `import` declaration in line 4 is *not* required if we always refer to class `Scanner` as `java.util.Scanner`, which includes the *full package name and class name*. This is known as the class's **fully qualified class name**. For example, line 12 could be written as

```
      java.util.Scanner input = new java.util.Scanner( System.in );
```

B.4 Instance Variables, *set* Methods and *get* Methods

In Appendix A, we declared all of an application's variables in the application's `main` method. Variables declared in the body of a particular method are known as **local variables** and can be used only in that method. When that method terminates, the values of its local variables are lost. Recall from Section 1.8 that an object has *attributes* that are carried with it as it's used in a program. Such attributes exist before a method is called on an object, while the method is executing and after the method completes execution.

A class normally consists of one or more methods that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class declaration. Such variables are called **fields** and are declared *inside* a class declaration but *outside* the bodies of the class's method declarations. When each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as an **instance variable**—each object (instance) of the class has a separate instance of the variable in memory. The example in this section demonstrates a `GradeBook` class that contains a `courseName` instance variable to represent a particular `GradeBook` object's course name.

`GradeBook` Class with an Instance Variable, a set *Method and a* get *Method*

In our next application (Figs. B.5–B.6), class `GradeBook` (Fig. B.5) maintains the course name as an instance variable so that it can be used or modified at any time during an application's execution. The class contains three methods—`setCourseName`, `getCourseName` and `displayMessage`. Method `setCourseName` stores a course name in a `GradeBook`. Method `getCourseName` obtains a `GradeBook`'s course name. Method `displayMessage`, which now specifies no parameters, still displays a welcome message that includes the course name; as you'll see, the method now obtains the course name by calling a method in the same class—`getCourseName`.

```
1  // Fig. B.5: GradeBook.java
2  // GradeBook class that contains a courseName instance variable
3  // and methods to set and get its value.
4
5  public class GradeBook
6  {
```

Fig. B.5 | `GradeBook` class that contains a `courseName` instance variable and methods to set and get its value. (Part 1 of 2.)

```
 7      private String courseName; // course name for this GradeBook
 8
 9      // method to set the course name
10      public void setCourseName( String name )
11      {
12         courseName = name; // store the course name
13      } // end method setCourseName
14
15      // method to retrieve the course name
16      public String getCourseName()
17      {
18         return courseName;
19      } // end method getCourseName
20
21      // display a welcome message to the GradeBook user
22      public void displayMessage()
23      {
24         // calls getCourseName to get the name of
25         // the course this GradeBook represents
26         System.out.printf( "Welcome to the grade book for\n%s!\n",
27            getCourseName() );
28      } // end method displayMessage
29   } // end class GradeBook
```

Fig. B.5 | `GradeBook` class that contains a `courseName` instance variable and methods to set and get its value. (Part 2 of 2.)

A typical instructor teaches more than one course, each with its own course name. Line 7 declares `courseName` as a variable of type `String`. Because the variable is declared *in* the body of the class but *outside* the bodies of the class's methods (lines 10–13, 16–19 and 22–28), line 7 is a declaration for an *instance variable*. Every instance (i.e., object) of class `GradeBook` contains one copy of each instance variable. For example, if there are two `GradeBook` objects, each object has its own copy of `courseName`. A benefit of making `courseName` an instance variable is that all the methods of the class (in this case, `GradeBook`) can manipulate any instance variables that appear in the class (in this case, `courseName`).

Access Modifiers `public` *and* `private`

Most instance-variable declarations are preceded with the keyword `private` (as in line 7). Like `public`, keyword **`private`** is an *access modifier. Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.* Thus, variable `courseName` can be used only in methods `setCourseName`, `getCourseName` and `displayMessage` of (every object of) class `GradeBook`.

Declaring instance variables with access modifier `private` is known as **data hiding** or information hiding. When a program creates (instantiates) an object of class `GradeBook`, variable `courseName` is *encapsulated* (hidden) in the object and can be accessed only by methods of the object's class. This prevents `courseName` from being modified accidentally by a class in another part of the program. In class `GradeBook`, methods `setCourseName` and `getCourseName` manipulate the instance variable `courseName`.

Software Engineering Observation B.1

Precede each field and method declaration with an access modifier. Generally, instance variables should be declared `private` and methods `public`. (It's appropriate to declare certain methods `private`, if they'll be accessed only by other methods of the class.)

Methods `setCourseName` *and* `getCourseName`

Method `setCourseName` (lines 10–13) does not return any data when it completes its task, so its return type is `void`. The method receives one parameter—`name`—which represents the course name that will be passed to the method as an argument. Line 12 assigns `name` to instance variable `courseName`.

Method `getCourseName` (lines 16–19) returns a particular `GradeBook` object's `courseName`. The method has an empty parameter list, so it does not require additional information to perform its task. The method specifies that it returns a `String`—this is the method's return type. When a method that specifies a return type other than `void` is called and completes its task, the method returns a *result* to its calling method. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you back a value that represents your balance. Similarly, when a statement calls method `getCourseName` on a `GradeBook` object, the statement expects to receive the `GradeBook`'s course name (in this case, a `String`, as specified in the method declaration's return type).

The **`return`** statement in line 18 passes the value of instance variable `courseName` back to the statement that calls method `getCourseName`. Consider, method `displayMessage`'s line 27, which calls method `getCourseName`. When the value is returned, the statement in lines 26–27 uses that value to output the course name. Similarly, if you have a method `square` that returns the square of its argument, you'd expect the statement

```
int result = square( 2 );
```

to return `4` from method `square` and assign `4` to the variable `result`. If you have a method `maximum` that returns the largest of three integer arguments, you'd expect the statement

```
int biggest = maximum( 27, 114, 51 );
```

to return `114` from method `maximum` and assign 114 to variable `biggest`.

The statements in lines 12 and 18 each use `courseName` *even though it was not declared in any of the methods.* We can use `courseName` in `GradeBook`'s methods because `courseName` is an instance variable of the class.

Method `displayMessage`

Method `displayMessage` (lines 22–28) does *not* return any data when it completes its task, so its return type is `void`. The method does *not* receive parameters, so the parameter list is empty. Lines 26–27 output a welcome message that includes the value of instance variable `courseName`, which is returned by the call to method `getCourseName` in line 27. Notice that one method of a class (`displayMessage` in this case) can call another method of the *same* class by using just the method name (`getCourseName` in this case).

`GradeBookTest` *Class That Demonstrates Class* `GradeBook`

Class `GradeBookTest` (Fig. B.6) creates one object of class `GradeBook` and demonstrates its methods. Line 14 creates a `GradeBook` object and assigns it to local variable `myGradeBook` of

```
 1  // Fig. B.6: GradeBookTest.java
 2  // Creating and manipulating a GradeBook object.
 3  import java.util.Scanner; // program uses Scanner
 4
 5  public class GradeBookTest
 6  {
 7     // main method begins program execution
 8     public static void main( String[] args )
 9     {
10        // create Scanner to obtain input from command window
11        Scanner input = new Scanner( System.in );
12
13        // create a GradeBook object and assign it to myGradeBook
14        GradeBook myGradeBook = new GradeBook();
15
16        // display initial value of courseName
17        System.out.printf( "Initial course name is: %s\n\n",
18           myGradeBook.getCourseName() );
19
20        // prompt for and read course name
21        System.out.println( "Please enter the course name:" );
22        String theName = input.nextLine(); // read a line of text
23        myGradeBook.setCourseName( theName ); // set the course name
24        System.out.println(); // outputs a blank line
25
26        // display welcome message after specifying course name
27        myGradeBook.displayMessage();
28     } // end main
29  } // end class GradeBookTest
```

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. B.6 | Creating and manipulating a `GradeBook` object.

type `GradeBook`. Lines 17–18 display the initial course name calling the object's `getCourseName` method. The first line of the output shows the name "`null`." *Unlike local variables, which are not automatically initialized, every field has a* ***default initial value****—a value provided by Java when you do not specify the field's initial value.* Thus, fields are *not* required to be explicitly initialized before they're used in a program—unless they must be initialized to values *other than* their default values. The default value for a field of type `String` (like `courseName` in this example) is `null`, which we say more about in Section B.5.

Line 21 prompts the user to enter a course name. Local `String` variable `theName` (declared in line 22) is initialized with the course name entered by the user, which is returned by the call to the `nextLine` method of the `Scanner` object `input`. Line 23 calls object `myGradeBook`'s `setCourseName` method and supplies `theName` as the method's argument. When the method is called, the argument's value is assigned to parameter `name` (line

10, Fig. B.5) of method setCourseName (lines 10–13, Fig. B.5). Then the parameter's value is assigned to instance variable courseName (line 12, Fig. B.5). Line 24 (Fig. B.6) skips a line in the output, then line 27 calls object myGradeBook's displayMessage method to display the welcome message containing the course name.

set *and* get *Methods*

A class's private fields can be manipulated *only* by the class's methods. So a **client of an object**—that is, any class that calls the object's methods—calls the class's public methods to manipulate the private fields of an object of the class. This is why the statements in method main (Fig. B.6) call the setCourseName, getCourseName and displayMessage methods on a GradeBook object. Classes often provide public methods to allow clients to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) private instance variables. The names of these methods need not begin with *set* or *get*, but this naming convention is recommended and is convention for special Java software components called JavaBeans, which can simplify programming in many Java integrated development environments (IDEs). The method that *sets* instance variable courseName in this example is called setCourseName, and the method that *gets* its value is called getCourseName.

B.5 Primitive Types vs. Reference Types

Java's types are divided into primitive types and **reference types**. The primitive types are boolean, byte, char, short, int, long, float and double. All nonprimitive types are reference types, so classes, which specify the types of objects, are reference types.

A primitive-type variable can store exactly one *value of its declared type* at a time. For example, an int variable can store one whole number (such as 7) at a time. When another value is assigned to that variable, its initial value is replaced. Primitive-type instance variables are *initialized by default*—variables of types byte, char, short, int, long, float and double are initialized to 0, and variables of type boolean are initialized to false. You can specify your own initial value for a primitive-type variable by assigning the variable a value in its declaration, as in

```
    private int numberOfStudents = 10;
```

Recall that local variables are *not* initialized by default.

Error-Prevention Tip B.1
An attempt to use an uninitialized local variable causes a compilation error.

Programs use variables of reference types (normally called **references**) to store the *locations* of objects in the computer's memory. Such a variable is said to **refer to an object** in the program. Objects that are referenced may each contain many instance variables. Line 14 of Fig. B.6 creates an object of class GradeBook, and the variable myGradeBook contains a reference to that GradeBook object. *Reference-type instance variables are initialized by default to the value* null—a reserved word that represents a "reference to nothing." This is why the first call to getCourseName in line 18 of Fig. B.6 returned null—the value of courseName had not been set, so the default initial value null was returned.

When you use an object of another class, a reference to the object is required to **invoke** (i.e., call) its methods. In the application of Fig. B.6, the statements in method main use the variable myGradeBook to send messages to the GradeBook object. These messages are calls to methods (like setCourseName and getCourseName) that enable the program to interact with the GradeBook object. For example, the statement in line 23 uses myGradeBook to send the setCourseName message to the GradeBook object. The message includes the argument that setCourseName requires to perform its task. The GradeBook object uses this information to set the courseName instance variable. Primitive-type variables do not refer to objects, so such variables cannot be used to invoke methods.

Software Engineering Observation B.2

A variable's declared type (e.g., int*,* double *or* GradeBook*) indicates whether the variable is of a primitive or a reference type. If a variable is* not *of one of the eight primitive types, then it's of a reference type.*

B.6 Initializing Objects with Constructors

As mentioned in Section B.4, when an object of class GradeBook (Fig. B.5) is created, its instance variable courseName is initialized to null by default. What if you want to provide a course name when you create a GradeBook object? Each class you declare can provide a special method called a constructor that can be used to initialize an object of a class when the object is created. In fact, Java *requires* a constructor call for *every* object that's created. Keyword new requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object. The call is indicated by the parentheses after the class name. A constructor *must* have the *same name* as the class. For example, line 14 of Fig. B.6 first uses new to create a GradeBook object. The empty parentheses after "new GradeBook" indicate a call to the class's constructor without arguments. By default, the compiler provides a **default constructor** with *no parameters* in any class that does *not* explicitly include a constructor. When a class has only the default constructor, its instance variables are initialized to their *default values*.

When you declare a class, you can provide your own constructor to specify custom initialization for objects of your class. For example, you might want to specify a course name for a GradeBook object when the object is created, as in

```
GradeBook myGradeBook =
   new GradeBook( "CS101 Introduction to Java Programming" );
```

In this case, the argument "CS101 Introduction to Java Programming" is passed to the GradeBook object's constructor and used to initialize the courseName. The preceding statement requires that the class provide a constructor with a String parameter. Figure B.7 contains a modified GradeBook class with such a constructor.

```
// Fig. B.7: GradeBook.java
// GradeBook class with a constructor to initialize the course name.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

```

Fig. B.7 | GradeBook class with a constructor to initialize the course name. (Part 1 of 2.)

```
 8       // constructor initializes courseName with String argument
 9       public GradeBook( String name ) // constructor name is class name
10       {
11          courseName = name; // initializes courseName
12       } // end constructor
13
14       // method to set the course name
15       public void setCourseName( String name )
16       {
17          courseName = name; // store the course name
18       } // end method setCourseName
19
20       // method to retrieve the course name
21       public String getCourseName()
22       {
23          return courseName;
24       } // end method getCourseName
25
26       // display a welcome message to the GradeBook user
27       public void displayMessage()
28       {
29          // this statement calls getCourseName to get the
30          // name of the course this GradeBook represents
31          System.out.printf( "Welcome to the grade book for\n%s!\n",
32             getCourseName() );
33       } // end method displayMessage
34    } // end class GradeBook
```

Fig. B.7 | `GradeBook` class with a constructor to initialize the course name. (Part 2 of 2.)

Lines 9–12 declare `GradeBook`'s constructor. Like a method, a constructor's parameter list specifies the data it requires to perform its task. When you create a new object (as we'll do in Fig. B.8), this data is placed in the *parentheses that follow the class name*. Line 9 of Fig. B.7 indicates that the constructor has a `String` parameter called `name`. The `name` passed to the constructor is assigned to instance variable `courseName` in line 11.

Figure B.8 initializes `GradeBook` objects using the constructor. Lines 11–12 create and initialize the `GradeBook` object `gradeBook1`. The `GradeBook` constructor is called with the argument `"CS101 Introduction to Java Programming"` to initialize the course name. The class instance creation expression in lines 11–12 returns a reference to the new object, which is assigned to the variable `gradeBook1`. Lines 13–14 repeat this process, this time passing the argument `"CS102 Data Structures in Java"` to initialize the course name for `gradeBook2`. Lines 17–20 use each object's `getCourseName` method to obtain the course names and show that they were initialized when the objects were created. The output confirms that each `GradeBook` maintains its own copy of instance variable `courseName`.

An important difference between constructors and methods is that constructors cannot return values, so they cannot specify a return type (not even `void`). Normally, constructors are declared `public`. If a class does not include a constructor, the class's instance variables are initialized to their default values. *If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.* Thus, we can no longer create a `GradeBook` object with `new GradeBook()` as we did in the earlier examples.

```java
// Fig. B.8: GradeBookTest.java
// GradeBook constructor used to specify the course name at the
// time each GradeBook object is created.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // create GradeBook object
      GradeBook gradeBook1 = new GradeBook(
         "CS101 Introduction to Java Programming" );
      GradeBook gradeBook2 = new GradeBook(
         "CS102 Data Structures in Java" );

      // display initial value of courseName for each GradeBook
      System.out.printf( "gradeBook1 course name is: %s\n",
         gradeBook1.getCourseName() );
      System.out.printf( "gradeBook2 course name is: %s\n",
         gradeBook2.getCourseName() );
   } // end main
} // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Fig. B.8 | GradeBook constructor used to specify the course name at the time each GradeBook object is created.

Constructors with Multiple Parameters
Sometimes you'll want to initialize objects with multiple data items. In Exercise B.11, we ask you to store the course name *and* the instructor's name in a GradeBook object. In this case, the GradeBook's constructor would be modified to receive two Strings, as in

```java
public GradeBook( String courseName, String instructorName )
```

and you'd call the GradeBook constructor as follows:

```java
GradeBook gradeBook = new GradeBook(
   "CS101 Introduction to Java Programming", "Sue Green" );
```

B.7 Floating-Point Numbers and Type double

We now depart temporarily from our GradeBook case study to declare an Account class that maintains the balance of a bank account. Most account balances are not whole numbers (such as 0, –22 and 1024). For this reason, class Account represents the account balance as a **floating-point number** (i.e., a number with a decimal point, such as 7.33, 0.0975 or 1000.12345). Java provides two primitive types for storing floating-point numbers in memory—float and double. They differ primarily in that double variables can store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point—also known as the number's **precision**) than float variables.

Floating-Point Number Precision and Memory Requirements

Variables of type **`float`** represent **single-precision floating-point numbers** and can represent up to *seven significant digits*. Variables of type **`double`** represent **double-precision floating-point numbers**. These require twice as much memory as `float` variables and provide *15 significant digits*—approximately double the precision of `float` variables. For the range of values required by most programs, variables of type `float` should suffice, but you can use `double` to "play it safe." In some applications, even `double` variables will be inadequate. Most programmers represent floating-point numbers with type `double`. In fact, Java treats all floating-point numbers you type in a program's source code (such as 7.33 and 0.0975) as `double` values by default. Such values in the source code are known as **floating-point literals**. See Appendix L for the ranges of values for `float`s and `double`s.

Although floating-point numbers are not always 100% precise, they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures. Owing to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more accurately. For this reason, we primarily use type `double` throughout the book. For precise floating-point numbers, Java provides class `BigDecimal` (package `java.math`).

Floating-point numbers also arise as a result of division. In conventional arithmetic, when we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.

`Account` Class with an Instance Variable of Type `double`

Our next application (Figs. B.9–B.10) contains a class named `Account` (Fig. B.9) that maintains the balance of a bank account. A typical bank services many accounts, each with its own balance, so line 7 declares an instance variable named `balance` of type `double`. It's an instance variable because it's declared in the body of the class but outside the class's method declarations (lines 10–16, 19–22 and 25–28). Every instance (i.e., object) of class `Account` contains its own copy of `balance`.

The class has a constructor and two methods. It's common for someone opening an account to deposit money immediately, so the constructor (lines 10–16) receives a parameter `initialBalance` of type `double` that represents the *starting balance*. Lines 14–15 ensure that `initialBalance` is greater than `0.0`. If so, `initialBalance`'s value is assigned to instance variable `balance`. Otherwise, `balance` remains at `0.0`—its default initial value.

```java
1  // Fig. B.9: Account.java
2  // Account class with a constructor to validate and
3  // initialize instance variable balance of type double.
4
5  public class Account
6  {
```

Fig. B.9 | `Account` class with a constructor to validate and initialize instance variable `balance` of type `double`. (Part 1 of 2.)

```
 7      private double balance; // instance variable that stores the balance
 8
 9      // constructor
10      public Account( double initialBalance )
11      {
12         // validate that initialBalance is greater than 0.0;
13         // if it is not, balance is initialized to the default value 0.0
14         if ( initialBalance > 0.0 )
15            balance = initialBalance;
16      } // end Account constructor
17
18      // credit (add) an amount to the account
19      public void credit( double amount )
20      {
21         balance = balance + amount; // add amount to balance
22      } // end method credit
23
24      // return the account balance
25      public double getBalance()
26      {
27         return balance; // gives the value of balance to the calling method
28      } // end method getBalance
29   } // end class Account
```

Fig. B.9 | `Account` class with a constructor to validate and initialize instance variable `balance` of type `double`. (Part 2 of 2.)

Method `credit` (lines 19–22) does *not* return any data when it completes its task, so its return type is `void`. The method receives one parameter named `amount`—a `double` value that will be added to the balance. Line 21 adds `amount` to the current value of `balance`, then assigns the result to `balance` (thus replacing the prior balance amount).

Method `getBalance` (lines 25–28) allows clients of the class (i.e., other classes that use this class) to obtain the value of a particular `Account` object's `balance`. The method specifies return type `double` and an empty parameter list.

Once again, the statements in lines 15, 21 and 27 use instance variable `balance` even though it was *not* declared in any of the methods. We can use `balance` in these methods because it's an instance variable of the class.

`AccountTest` Class to Use Class `Account`

Class `AccountTest` (Fig. B.10) creates two `Account` objects (lines 10–11) and initializes them with `50.00` and `-7.53`, respectively. Lines 14–17 output the balance in each `Account` by calling the `Account`'s `getBalance` method. When method `getBalance` is called for `account1` from line 15, the value of `account1`'s balance is returned from line 27 of Fig. B.9 and displayed by the `System.out.printf` statement (Fig. B.10, lines 14–15). Similarly, when method `getBalance` is called for `account2` from line 17, the value of the `account2`'s balance is returned from line 27 of Fig. B.9 and displayed by the `System.out.printf` statement (Fig. B.10, lines 16–17). The balance of `account2` is `0.00`, because the constructor ensured that the account could *not* begin with a negative balance. The value is output by `printf` with the format specifier `%.2f`. The format specifier **`%f`** is used to output values of type `float` or `double`. The `.2` between `%` and `f` represents the number of decimal places (2) that should be output to the right of the decimal point in the floating-point number—also known as the number's **precision**. Any floating-point value output with `%.2f` will be rounded to the hundredths position—for example, 123.457 would be rounded to 123.46, 27.333 would be rounded to 27.33 and 123.455 would be rounded to 123.46.

```java
// Fig. B.10: AccountTest.java
// Inputting and outputting floating-point numbers with Account objects.
import java.util.Scanner;

public class AccountTest
{
   // main method begins execution of Java application
   public static void main( String[] args )
   {
      Account account1 = new Account( 50.00 ); // create Account object
      Account account2 = new Account( -7.53 ); // create Account object

      // display initial balance of each object
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n\n",
         account2.getBalance() );

      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );
      double depositAmount; // deposit amount read from user

      System.out.print( "Enter deposit amount for account1: " ); // prompt
      depositAmount = input.nextDouble(); // obtain user input
      System.out.printf( "\nadding %.2f to account1 balance\n\n",
         depositAmount );
      account1.credit( depositAmount ); // add to account1 balance

      // display balances
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n\n",
         account2.getBalance() );

      System.out.print( "Enter deposit amount for account2: " ); // prompt
      depositAmount = input.nextDouble(); // obtain user input
      System.out.printf( "\nadding %.2f to account2 balance\n\n",
         depositAmount );
      account2.credit( depositAmount ); // add to account2 balance

      // display balances
      System.out.printf( "account1 balance: $%.2f\n",
         account1.getBalance() );
      System.out.printf( "account2 balance: $%.2f\n",
         account2.getBalance() );
   } // end main
} // end class AccountTest
```

Fig. B.10 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 2.)

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Fig. B.10 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 2.)

Line 21 declares local variable `depositAmount` to store each deposit amount entered by the user. Unlike the instance variable `balance` in class `Account`, local variable `depositAmount` in `main` is *not* initialized to 0.0 by default. However, this variable does not need to be initialized here, because its value will be determined by the user's input.

Line 23 prompts the user to enter a deposit amount for `account1`. Line 24 obtains the input from the user by calling `Scanner` object `input`'s **`nextDouble`** method, which returns a `double` value entered by the user. Lines 25–26 display the deposit amount. Line 27 calls object `account1`'s `credit` method and supplies `depositAmount` as the method's argument. When the method is called, the argument's value is assigned to parameter `amount` (line 19 of Fig. B.9) of method `credit` (lines 19–22 of Fig. B.9); then method `credit` adds that value to the `balance` (line 21 of Fig. B.9). Lines 30–33 (Fig. B.10) output the balances of both `Accounts` again to show that only `account1`'s balance changed.

Line 35 prompts the user to enter a deposit amount for `account2`. Line 36 obtains the input from the user by calling `Scanner` object `input`'s `nextDouble` method. Lines 37–38 display the deposit amount. Line 39 calls object `account2`'s `credit` method and supplies `depositAmount` as the method's argument; then method `credit` adds that value to the balance. Finally, lines 42–45 output the balances of both `Accounts` again to show that only `account2`'s balance changed.

B.8 Wrap-Up

In this appendix, you learned how to declare instance variables of a class to maintain data for each object of the class, and how to declare methods that operate on that data. You learned how to call a method to tell it to perform its task and how to pass information to methods as arguments. You learned the difference between a local variable of a method and an instance variable of a class and that only instance variables are initialized automatically. You also learned how to use a class's constructor to specify the initial values for an object's instance variables. Finally, you learned about floating-point numbers—how to store them with variables of primitive type `double`, how to input them with a `Scanner` object and how to format them with `printf` and format specifier `%f` for display purposes. In the next appendix we begin our introduction to control statements, which specify the order in which a program's actions are performed. You'll use these in your methods to specify how they should perform their tasks.

Self-Review Exercises

B.1 Fill in the blanks in each of the following:

a) A ________ method is a special method that you can call without first creating an object of the class in which it is declared.
b) Keyword ________ in a class declaration is followed immediately by the class's name.
c) Keyword ________ requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object.
d) Each parameter must specify both a(n) ________ and a(n) ________.
e) By default, classes that are compiled in the same directory are considered to be in the same package, known as the ________.
f) When each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as a(n) ________.
g) Java provides two primitive types for storing floating-point numbers in memory: ________ and ________.
h) Variables of type `double` represent ________ floating-point numbers.
i) `Scanner` method ________ returns a `double` value.
j) Keyword `public` is an access ________.
k) Return type ________ indicates that a method will not return a value.
l) `Scanner` method ________ reads characters until it encounters a newline character, then returns those characters as a `String`.
m) Class `String` is in package ________.
n) A(n) ________ is not required if you always refer to a class with its fully qualified class name.
o) A(n) ________ is a number with a decimal point, such as 7.33, 0.0975 or 1000.12345.
p) Variables of type `float` represent ________ floating-point numbers.
q) The format specifier ________ is used to output values of type `float` or `double`.
r) Types in Java are divided into two categories—________ types and ________ types.

B.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) By convention, method names begin with an uppercase first letter, and all subsequent words in the name begin with a capital first letter.
b) An `import` declaration is not required when one class in a package uses another in the same package.
c) Empty parentheses following a method name in a method declaration indicate that the method does not require any parameters to perform its task.
d) Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.
e) A primitive-type variable can be used to invoke a method.
f) The compiler provides a default constructor with no parameters in any class that does not explicitly include a constructor.
g) Every method's body is delimited by left and right braces (`{` and `}`).
h) Primitive-type local variables are initialized by default.
i) Reference-type instance variables are initialized by default to the value `null`.
j) Any class that contains `public static void main( String[] args )` can be used to execute an application.
k) The argument types in the method call need not be consistent with the types of corresponding parameters in the method's declaration.
l) Floating-point values that appear in source code are known as floating-point literals and are type `float` by default.

B.3 Explain the `public` and `private` access modifiers in brief.

B.4 Explain the purpose of a method parameter. What is the difference between a parameter and an argument?

Answers to Self-Review Exercises

B.1 a) `static`. b) `class`. c) `new`. d) type, name. e) default package. f) instance variable. g) `float`, `double`. h) double-precision. i) `nextDouble`. j) modifier. k) `void`. l) `nextLine`. m) `java.lang`. n) `import` declaration. o) floating-point number. p) single-precision. q) `%f`. r) primitive, reference.

B.2 a) False. By convention, method names begin with a lowercase first letter and all subsequent words in the name begin with a capital first letter. b) True. c) True. d) True. e) False. A primitive-type variable cannot be used to invoke a method—a reference to an object is required to invoke the object's methods. f) True. g) True. h) False. Primitive-type instance variables are initialized by default. Each local variable must explicitly be assigned a value. i) True. j) True. k) False. The argument types in the method call must be consistent with the types of corresponding parameters in the method's declaration. l) False. Such literals are of type `double` by default.

B.3 As the names suggest, variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared, whereas the variables or methods declared with access modifier `public` are "accessible to the public"—they can be accessed from methods of other classes.

B.4 A parameter represents additional information that a method requires to perform its task. Each parameter required by a method is specified in the method's declaration. An argument is the actual value for a method parameter. When a method is called, the argument values are passed to the corresponding parameters of the method so that it can perform its task.

Exercises

B.5 *(Class* ***Scanner****)* What's the purpose of class `Scanner`? Explain how to make use of its various features.

B.6 *(Primitive and Reference Types)* What are primitive and reference data types in Java? Explain the differences between them.

B.7 *(Static Variables)* Explain the purpose of a `static` variable.

B.8 *(Using Classes Without Importing Them)* Most classes need to be imported before they can be used in an application. Why is every application allowed to use classes `System` and `String` without first importing them?

B.9 *(App Compilation)* Briefly explain how to compile an application with multiple classes using an example.

B.10 *(Imports Declarations)* What is an import declaration? Explain its usage in java programming with an example.

B.11 *(Modified* ***GradeBook*** *Class)* Modify class `GradeBook` (Fig. B.7) as follows:

a) Include a `String` instance variable that represents the name of the course's instructor.
b) Provide a *set* method to change the instructor's name and a *get* method to retrieve it.
c) Modify the constructor to specify two parameters—one for the course name and one for the instructor's name.
d) Modify method `displayMessage` to output the welcome message and course name, followed by `"This course is presented by: "` and the instructor's name.

Use your modified class in a test application that demonstrates the class's new capabilities.

B.12 *(Modified* ***Account*** *Class)* Modify class `Account` (Fig. B.9) to provide a method called `debit` that withdraws money from an `Account`. Ensure that the debit amount does not exceed the

Account's balance. If it does, the balance should be left unchanged and the method should print a message indicating "Debit amount exceeded account balance." Modify class AccountTest (Fig. B.10) to test method debit.

B.13 *(Invoice Class)* Create a class called Invoice that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four pieces of information as instance variables—a part number (type String), a part description (type String), a quantity of the item being purchased (type int) and a price per item (double). Your class should have a constructor that initializes the four instance variables. Provide a *set* and a *get* method for each instance variable. In addition, provide a method named getInvoiceAmount that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a double value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0.0. Write a test application named InvoiceTest that demonstrates class Invoice's capabilities.

B.14 *(Employee Class)* Create a class called Employee that includes three instance variables—a first name (type String), a last name (type String) and a monthly salary (double). Provide a constructor that initializes the three instance variables. Provide a *set* and a *get* method for each instance variable. If the monthly salary is not positive, do not set its value. Write a test application named EmployeeTest that demonstrates class Employee's capabilities. Create two Employee objects and display each object's *yearly* salary. Then give each Employee a 10% raise and display each Employee's yearly salary again.

B.15 *(Date Class)* Create a class called Date that includes three instance variables—a month (type int), a day (type int) and a year (type int). Provide a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a *set* and a *get* method for each instance variable. Provide a method displayDate that displays the month, day and year separated by forward slashes (/). Write a test application named DateTest that demonstrates class Date's capabilities.

C Control Statements

Objectives

In this appendix you'll:

- Learn basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` and `if`...`else` selection statements to choose among alternative actions.
- Use the `while` repetition statement to execute statements in a program repeatedly.
- Use counter-controlled repetition and sentinel-controlled repetition.
- Use the compound assignment, increment and decrement operators.
- Learn the essentials of counter-controlled repetition.
- Use the `for` and `do`...`while` repetition statements to execute statements in a program repeatedly.
- Implement multiple selection using the `switch` statement.
- Use the `break` and `continue` statements .
- Use the logical operators in conditional expressions.

Outline

C.1 Introduction

In this appendix, we discuss the theory and principles of structured programming. The concepts presented here are crucial in building classes and manipulating objects. We introduce Java's compound assignment, increment and decrement operators, and we discuss the portability of Java's primitive types. We demonstrate Java's `for`, `do…while` and `switch` statements. Through a series of short examples using `while` and `for`, we explore the essentials of counter-controlled repetition. We create a version of class `GradeBook` that uses a `switch` statement to count the number of A, B, C, D and F grade equivalents in a set of numeric grades entered by the user. We introduce the `break` and `continue` program-control statements. We discuss Java's logical operators, which enable you to use more complex conditional expressions in control statements.

C.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of

1. the **actions** to execute and
2. the **order** in which these actions execute

is called an **algorithm**. Correctly specifying the order in which the actions execute is important.

Consider the "rise-and-shine algorithm" followed by one executive for getting out of bed and going to work: (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work. In this case, our executive shows up for work soaking wet. Specifying the order in which statements (actions) execute in a program is called **program control**. This appendix investigates program control using Java's **control statements**.

C.3 Pseudocode

Pseudocode is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of Java programs. Pseudocode is similar to everyday English—it's convenient and user friendly, but it's not an actual computer programming language.

Pseudocode does not execute on computers. Rather, it helps you "think out" a program before attempting to write it in a programming language, such as Java. Pseudocode normally describes only statements representing the actions that occur after you convert a program from pseudocode to Java and the program is run on a computer. Such actions might include input, output or calculations.

C.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they're written. This process is called **sequential execution**. Various Java statements, which we'll soon discuss, enable you to specify that the next statement to execute is *not* necessarily the *next* one in sequence. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the **goto statement** (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program. The term **structured programming** became almost synonymous with "goto elimination." [*Note:* Java does *not* have a goto statement; however, the word goto is *reserved* by Java and should *not* be used as an identifier in programs.]

Research had demonstrated that programs could be written *without* any goto statements. The challenge of the era for programmers was to shift their styles to "goto-less programming." Not until the 1970s did most programmers start taking structured programming seriously. The results were impressive. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug free in the first place.

Researchers demonstrated that all programs could be written in terms of only three control structures—the **sequence structure**, the **selection structure** and the **repetition structure**. When we introduce Java's control structure implementations, we'll refer to them in the terminology of the *Java Language Specification* as "control statements."

Sequence Structure in Java

The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they're written—that is, in sequence. Java lets you have as many actions as you want in a sequence structure. As we'll soon see, anywhere a single action may be placed, we may place several actions in sequence.

Selection Statements in Java

Java has three types of **selection statements**. The `if` statement either performs (selects) an action, if a condition is true, or skips it, if the condition is false. The `if...else` statement performs an action if a condition is true and performs a different action if the condition is false. The `switch` statement performs one of many different actions, depending on the value of an expression.

The `if` statement is a **single-selection statement** because it selects or ignores a *single* action (or, as we'll soon see, a *single group of actions*). The `if...else` statement is called a **double-selection statement** because it selects between *two different actions* (or *groups of actions*). The `switch` statement is called a **multiple-selection statement** because it selects among *many different actions* (or *groups of actions*).

Repetition Statements in Java

Java provides three **repetition statements** (also called **looping statements**) that enable programs to perform statements repeatedly as long as a condition (called the **loop-continuation condition**) remains true. The repetition statements are the `while`, `do...while` and `for` statements. The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially false, the action (or group of actions) will not execute. The `do...while` statement performs the action (or group of actions) in its body *one or more* times. The words `if`, `else`, `switch`, `while`, `do` and `for` are Java keywords.

C.5 `if` Single-Selection Statement

Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

determines whether the condition "student's grade is greater than or equal to 60" is true. If so, "Passed" is printed, and the next pseudocode statement in order is "performed." If the condition is false, the *Print* statement is ignored, and the next pseudocode statement in order is performed.

The preceding pseudocode *If* statement easily may be converted to the Java statement

```java
if ( studentGrade >= 60 )
   System.out.println( "Passed" );
```

C.6 `if...else` Double-Selection Statement

The `if` single-selection statement performs an indicated action only when the condition is `true`; otherwise, the action is skipped. The **`if...else` double-selection statement** allows you to specify an action to perform when the condition is true and a different action when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

prints “Passed” if the student’s grade is greater than or equal to 60, but prints “Failed” if it’s less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is “performed.”

The preceding *If…Else* pseudocode statement can be written in Java as

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
   System.out.println( "Failed" );
```

Conditional Operator (`?:`)

Java provides the **conditional operator** (**`?:`**) that can be used in place of an `if…else` statement. This is Java’s only **ternary operator** (operator that takes three operands). Together, the operands and the `?:` symbol form a **conditional expression.** The first operand (to the left of the `?`) is a **`boolean` expression** (i.e., a condition that evaluates to a `boolean` value—**`true`** or **`false`**), the second operand (between the `?` and `:`) is the value of the conditional expression if the `boolean` expression is `true` and the third operand (to the right of the `:`) is the value of the conditional expression if the `boolean` expression evaluates to `false`. For example, the statement

```
System.out.println( studentGrade >= 60 ? "Passed" : "Failed" );
```

prints the value of `println`’s conditional-expression argument. The conditional expression in this statement evaluates to the string `"Passed"` if the `boolean` expression `studentGrade >= 60` is true and to the string `"Failed"` if it’s false. Thus, this statement with the conditional operator performs essentially the same function as the `if…else` statement shown earlier in this section. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses.

Nested `if…else` Statements

A program can test multiple cases by placing `if…else` statements inside other `if…else` statements to create **nested `if…else` statements**. For example, the following pseudocode represents a nested `if…else` that prints `A` for exam grades greater than or equal to 90, `B` for grades 80 to 89, `C` for grades 70 to 79, `D` for grades 60 to 69 and `F` for all other grades:

```
If student’s grade is greater than or equal to 90
    Print “A”
else
    If student’s grade is greater than or equal to 80
        Print “B”
    else
        If student’s grade is greater than or equal to 70
            Print “C”
        else
            If student’s grade is greater than or equal to 60
                Print “D”
            else
                Print “F”
```

This pseudocode may be written in Java as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else
   if ( studentGrade >= 80 )
      System.out.println( "B" );
   else
      if ( studentGrade >= 70 )
         System.out.println( "C" );
      else
         if ( studentGrade >= 60 )
            System.out.println( "D" );
         else
            System.out.println( "F" );
```

If variable `studentGrade` is greater than or equal to 90, the first four conditions in the nested `if...else` statement will be true, but only the statement in the `if` part of the first `if...else` statement will execute. After that statement executes, the `else` part of the "outermost" `if...else` statement is skipped. Many programmers prefer to write the preceding nested `if...else` statement as

```
if ( studentGrade >= 90 )
   System.out.println( "A" );
else if ( studentGrade >= 80 )
   System.out.println( "B" );
else if ( studentGrade >= 70 )
   System.out.println( "C" );
else if ( studentGrade >= 60 )
   System.out.println( "D" );
else
   System.out.println( "F" );
```

The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form avoids deep indentation of the code to the right.

Blocks

The `if` statement normally expects only one statement in its body. To include several statements in the body of an `if` (or the body of an `else` for an `if...else` statement), enclose the statements in braces. Statements contained in a pair of braces form a **block**. A block can be placed anywhere in a program that a single statement can be placed. The following example includes a block in the `else` part of an `if...else` statement:

```
if ( grade >= 60 )
   System.out.println( "Passed" );
else
{
   System.out.println( "Failed" );
   System.out.println( "You must take this course again." );
}
```

In this case, if `grade` is less than 60, the program executes *both* statements in the body of the `else` and prints

```
Failed
You must take this course again.
```

Note the braces surrounding the two statements in the `else` clause. These braces are important. Without the braces, the statement

```
System.out.println( "You must take this course again." );
```

would be outside the body of the `else` part of the `if...else` statement and would execute *regardless* of whether the grade was less than 60.

Syntax errors (e.g., when one brace in a block is left out of the program) are caught by the compiler. A **logic error** (e.g., when both braces in a block are left out of the program) has its effect at execution time. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing but causes it to produce incorrect results.

C.7 `while` Repetition Statement

As an example of Java's **`while` repetition statement**, consider a program segment that finds the first power of 3 larger than 100. Suppose that the `int` variable `product` is initialized to `3`. After the following `while` statement executes, `product` contains the result:

```
while ( product <= 100 )
   product = 3 * product;
```

When this `while` statement begins execution, the value of variable `product` is 3. Each iteration of the `while` statement multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When variable `product` becomes 243, the `while`-statement condition—`product <= 100`—becomes false. This terminates the repetition, so the final value of `product` is 243. At this point, program execution continues with the next statement after the `while` statement .

Common Programming Error C.1

*Not providing in the body of a `while` statement an action that eventually causes the condition in the `while` to become false normally results in a logic error called an **infinite loop** (the loop never terminates).*

C.8 Case Study: Counter-Controlled Repetition

To illustrate how algorithms are developed, we modify the `GradeBook` class of Appendix B to solve two variations of a problem that averages student grades. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students. The algorithm for solving this problem on a computer must input each grade, keep track of the total of all grades input, perform the averaging calculation and print the result.

Pseudocode Algorithm with Counter-Controlled Repetition

Let's use pseudocode to list the actions to execute and specify the order in which they should execute. We use **counter-controlled repetition** to input the grades one at a time. This technique uses a variable called a **counter** (or **control variable**) to control the number of times a set of statements will execute. In this example, repetition terminates when the counter exceeds 10. This section presents a fully developed pseudocode algorithm (Fig. C.1) and a version of class `GradeBook` (Fig. C.2) that implements the algorithm in a Java method. We then present an application (Fig. C.3) that demonstrates the algorithm in action.

Note the references in the algorithm of Fig. C.1 to a total and a counter. A **total** is a variable used to accumulate the sum of several values. A counter is a variable used to count—in this case, the grade counter indicates which of the 10 grades is about to be entered by the user. Variables used to store totals are normally initialized to zero before being used in a program.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Prompt the user to enter the next grade
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. C.1 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

Implementing Counter-Controlled Repetition in Class `GradeBook`

Class `GradeBook` (Fig. C.2) contains a constructor (lines 11–14) that assigns a value to the class's instance variable `courseName` (declared in line 8). Lines 17–20, 23–26 and 29–34 declare methods `setCourseName`, `getCourseName` and `displayMessage`, respectively. Lines 37–66 declare method `determineClassAverage`, which implements the class-averaging algorithm described by the pseudocode in Fig. C.1.

Line 40 declares and initializes `Scanner` variable `input`, which is used to read values entered by the user. Lines 42–45 declare local variables `total`, `gradeCounter`, `grade` and `average` to be of type `int`. Variable `grade` stores the user input.

```
1  // Fig. C.2: GradeBook.java
2  // GradeBook class that solves the class-average problem using
3  // counter-controlled repetition.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class GradeBook
7  {
8     private String courseName; // name of course this GradeBook represents
```

Fig. C.2 | `GradeBook` class that solves the class-average problem using counter-controlled repetition. (Part 1 of 3.)

```java

   // constructor initializes courseName
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // determine class average based on 10 grades entered by user
   public void determineClassAverage()
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int total; // sum of grades entered by user
      int gradeCounter; // number of the grade to be entered next
      int grade; // grade value entered by user
      int average; // average of grades

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 1; // initialize loop counter

      // processing phase uses counter-controlled repetition
      while ( gradeCounter <= 10 ) // loop 10 times
      {
         System.out.print( "Enter grade: " ); // prompt
         grade = input.nextInt(); // input next grade
         total = total + grade; // add grade to total
         gradeCounter = gradeCounter + 1; // increment counter by 1
      } // end while

```

Fig. C.2 | GradeBook class that solves the class-average problem using counter-controlled repetition. (Part 2 of 3.)

```
60        // termination phase
61        average = total / 10; // integer division yields integer result
62
63        // display total and average of grades
64        System.out.printf( "\nTotal of all 10 grades is %d\n", total );
65        System.out.printf( "Class average is %d\n", average );
66     } // end method determineClassAverage
67  } // end class GradeBook
```

Fig. C.2 | `GradeBook` class that solves the class-average problem using counter-controlled repetition. (Part 3 of 3.)

The declarations (in lines 42–45) appear in the body of method `determineClassAverage`. A local variable's declaration must appear *before* the variable is used in that method. A local variable cannot be accessed outside the method in which it's declared.

The assignments (in lines 48–49) initialize `total` to `0` and `gradeCounter` to `1`. Line 52 indicates that the `while` statement should continue looping (also called **iterating**) as long as `gradeCounter`'s value is less than or equal to 10. While this condition remains true, the `while` statement repeatedly executes the statements between the braces that delimit its body (lines 54–57).

Line 54 displays the prompt `"Enter grade: "`. Line 55 reads the grade entered by the user and assigns it to variable `grade`. Then line 56 adds the new `grade` entered by the user to the `total` and assigns the result to `total`, which replaces its previous value.

Line 57 adds `1` to `gradeCounter` to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing `gradeCounter` eventually causes it to exceed 10. Then the loop terminates, because its condition (line 52) becomes false.

When the loop terminates, line 61 performs the averaging calculation and assigns its result to the variable `average`. Line 64 uses `System.out`'s `printf` method to display the text `"Total of all 10 grades is "` followed by variable `total`'s value. Line 65 then uses `printf` to display the text `"Class average is "` followed by variable `average`'s value. After reaching line 66, method `determineClassAverage` returns control to the calling method (i.e., `main` in `GradeBookTest` of Fig. C.3).

Class `GradeBookTest`

Class `GradeBookTest` (Fig. C.3) creates an object of class `GradeBook` (Fig. C.2) and demonstrates its capabilities. Lines 10–11 of Fig. C.3 create a new `GradeBook` object and assign it to variable `myGradeBook`. The `String` in line 11 is passed to the `GradeBook` constructor (lines 11–14 of Fig. C.2). Line 13 calls `myGradeBook`'s `displayMessage` method to display a welcome message to the user. Line 14 then calls `myGradeBook`'s `determineClassAverage` method to allow the user to enter 10 grades, for which the method then calculates and prints the average—the method performs the algorithm shown in Fig. C.1.

```
1  // Fig. C.3: GradeBookTest.java
2  // Create GradeBook object and invoke its determineClassAverage method.
3
```

Fig. C.3 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. C.2) and invokes its `determineClassAverage` method. (Part 1 of 2.)

```
 4  public class GradeBookTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // create GradeBook object myGradeBook and
 9        // pass course name to constructor
10        GradeBook myGradeBook = new GradeBook(
11           "CS101 Introduction to Java Programming" );
12
13        myGradeBook.displayMessage(); // display welcome message
14        myGradeBook.determineClassAverage(); // find average of 10 grades
15     } // end main
16  } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. C.3 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. C.2) and invokes its `determineClassAverage` method. (Part 2 of 2.)

Notes on Integer Division and Truncation

The averaging calculation performed by method `determineClassAverage` in response to the method call at line 14 in Fig. C.3 produces an integer result. The program's output indicates that the sum of the grade values in the sample execution is 846, which, when divided by 10, should yield the floating-point number 84.6. However, the result of the calculation `total / 10` (line 61 of Fig. C.2) is the integer 84, because `total` and `10` are both integers. Dividing two integers results in **integer division**—any fractional part of the calculation is lost (i.e., **truncated**).

C.9 Case Study: Sentinel-Controlled Repetition

Let's generalize Section C.8's class-average problem. Consider the following problem:

> *Develop a class-averaging program that processes grades for an arbitrary number of students each time it's run.*

In the previous class-average example, the problem statement specified the number of students, so the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter during the program's execution. The program must process an arbitrary number of grades. How can it determine when to stop reading grades from the user? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate "end of data entry." The user enters grades until all legitimate grades have been entered. The user then types the sentinel value to indicate that no more grades will be entered. **Sentinel-controlled repetition** is often called **indefinite repetition** because the number of repetitions is *not* known before the loop begins executing.

Clearly, a sentinel value must be chosen that cannot be confused with an acceptable input value. Grades on a quiz are nonnegative integers, so –1 is an acceptable sentinel value for this problem. Thus, a run of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89; since –1 is the sentinel value, it should *not* enter into the averaging calculation. The complete pseudocode for the class-average problem is shown in Fig. C.4.

```
Initialize total to zero
Initialize counter to zero

Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print "No grades were entered"
```

Fig. C.4 | Class-average problem pseudocode algorithm with sentinel-controlled repetition.

Implementing Sentinel-Controlled Repetition in Class `GradeBook`

Figure C.5 shows the Java class `GradeBook` containing method `determineClassAverage` that implements the pseudocode algorithm of Fig. C.4. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point—in other words, a real (i.e., floating-point) number. The type `int` cannot represent such a number, so this class uses type `double` to do so.

```java
// Fig. C.5: GradeBook.java
// GradeBook class that solves the class-average problem using
// sentinel-controlled repetition.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents

   // constructor initializes courseName
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // determine the average of an arbitrary number of grades
   public void determineClassAverage()
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      int total; // sum of grades
      int gradeCounter; // number of grades entered
      int grade; // grade value
      double average; // number with decimal point for average

      // initialization phase
      total = 0; // initialize total
      gradeCounter = 0; // initialize loop counter

```

Fig. C.5 | GradeBook class that solves the class-average problem using sentinel-controlled repetition. (Part 1 of 2.)

```
51         // processing phase
52         // prompt for input and read grade from user
53         System.out.print( "Enter grade or -1 to quit: " );
54         grade = input.nextInt();
55
56         // loop until sentinel value read from user
57         while ( grade != -1 )
58         {
59            total = total + grade; // add grade to total
60            gradeCounter = gradeCounter + 1; // increment counter
61
62            // prompt for input and read next grade from user
63            System.out.print( "Enter grade or -1 to quit: " );
64            grade = input.nextInt();
65         } // end while
66
67         // termination phase
68         // if user entered at least one grade...
69         if ( gradeCounter != 0 )
70         {
71            // calculate average of all grades entered
72            average = (double) total / gradeCounter;
73
74            // display total and average (with two digits of precision)
75            System.out.printf( "\nTotal of the %d grades entered is %d\n",
76               gradeCounter, total );
77            System.out.printf( "Class average is %.2f\n", average );
78         } // end if
79         else // no grades were entered, so output appropriate message
80            System.out.println( "No grades were entered" );
81      } // end method determineClassAverage
82   } // end class GradeBook
```

Fig. C.5 | `GradeBook` class that solves the class-average problem using sentinel-controlled repetition. (Part 2 of 2.)

In this example, we see that control statements may be *stacked* on top of one another (in sequence). The `while` statement (lines 57–65) is followed in sequence by an `if...else` statement (lines 69–80). Much of the code in this program is identical to that in Fig. C.2, so we concentrate on the new concepts.

Line 45 declares `double` variable `average`, which allows us to store the class average as a floating-point number. Line 49 initializes `gradeCounter` to `0`, because no grades have been entered yet. To keep an accurate record of the number of grades entered, the program increments `gradeCounter` only when the user enters a valid grade.

Program Logic for Sentinel-Controlled Repetition vs. Counter-Controlled Repetition

Compare the program logic for sentinel-controlled repetition in this application with that for counter-controlled repetition in Fig. C.2. In counter-controlled repetition, each iteration of the `while` statement (e.g., lines 52–58 of Fig. C.2) reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads the first value (lines 53–54 of Fig. C.5) before reaching the `while`. This value determines whether the program's flow of control should enter the body of the `while`. If the condition of the `while` is false, the user entered the sentinel value, so the body of the `while` does not execute (i.e., no grades were entered). If, on the other hand, the condition is true, the body begins execution, and the loop adds the `grade` value to the `total` (line 59). Then lines 63–64 in the loop body input the next value from the user. Next, program control reaches the closing right brace of the loop body at line 65, so execution continues with the test of the `while`'s condition (line 57). The condition uses the most recent `grade` input by the user to determine whether the loop body should execute again. The value of variable `grade` is always input from the user immediately before the program tests the `while` condition. This allows the program to determine whether the value just input is the sentinel value *before* the program processes that value (i.e., adds it to the `total`). If the sentinel value is input, the loop terminates, and the program does not add –1 to the `total`.

After the loop terminates, the `if...else` statement at lines 69–80 executes. The condition at line 69 determines whether any grades were input. If none were input, the `else` part (lines 79–80) of the `if...else` statement executes and displays the message `"No grades were entered"` and the method returns control to the calling method.

Explicitly and Implicitly Converting Between Primitive Types

If at least one grade was entered, line 72 of Fig. C.5 calculates the average of the grades. Recall from Fig. C.2 that integer division yields an integer result. Even though variable `average` is declared as a `double` (line 45), the calculation

```
    average = total / gradeCounter;
```

loses the fractional part of the quotient *before* the result of the division is assigned to `average`. This occurs because `total` and `gradeCounter` are *both* integers, and integer division yields an integer result. To perform a floating-point calculation with integer values, we must temporarily treat these values as floating-point numbers for use in the calculation. Java provides the **unary cast operator** to accomplish this task. Line 72 uses the **`(double)`** cast operator—a unary operator—to create a *temporary* floating-point copy of its operand `total` (which appears to the right of the operator). Using a cast operator in this manner is called **explicit conversion** or **type casting**. The value stored in `total` is still an integer.

The calculation now consists of a floating-point value (the temporary `double` version of `total`) divided by the integer `gradeCounter`. Java knows how to evaluate only arithmetic expressions in which the operands' types are *identical*. To ensure that the operands are of the same type, Java performs an operation called **promotion** (or **implicit conversion**) on selected operands. For example, in an expression containing values of the types `int` and `double`, the `int` values are promoted to `double` values for use in the expression. In this example, the value of `gradeCounter` is promoted to type `double`, then the floating-point division is performed and the result of the calculation is assigned to `average`. As long as the `(double)` cast operator is applied to *any* variable in the calculation, the calculation will yield a `double` result.

A cast operator is formed by placing parentheses around any type's name. The operator is a **unary operator** (i.e., an operator that takes only one operand). Java also supports unary versions of the plus (+) and minus (–) operators, so you can write expressions like `-7` or `+5`. Cast operators associate from right to left and have the same precedence as other unary operators, such as unary `+` and unary `-`. (See the operator precedence chart in Appendix K.)

Line 77 displays the class average. In this example, we display the class average rounded to the nearest hundredth. The format specifier `%.2f` in `printf`'s format control string indicates that variable `average`'s value should be displayed with two digits of precision to the right of the decimal point—indicated by `.2` in the format specifier. The three grades entered during the sample execution of class `GradeBookTest` (Fig. C.6) total 257, which yields the average 85.666666…. Method `printf` uses the precision in the format specifier to round the value to the specified number of digits. In this program, the average is rounded to the hundredths position and is displayed as `85.67`.

```java
// Fig. C.6: GradeBookTest.java
// Create GradeBook object and invoke its determineClassAverage method.

public class GradeBookTest
{
   public static void main( String[] args )
   {
      // create GradeBook object myGradeBook and
      // pass course name to constructor
      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming" );

      myGradeBook.displayMessage(); // display welcome message
      myGradeBook.determineClassAverage(); // find average of grades
   } // end main
} // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Fig. C.6 | `GradeBookTest` class creates an object of class `GradeBook` (Fig. C.5) and invokes its `determineClassAverage` method.

C.10 Case Study: Nested Control Statements

We've seen that control statements can be stacked on top of one another (in sequence). In this case study, we examine the only other structured way control statements can be connected— **nesting** one control statement within another.

Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam.

The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.*
2. *Count the number of test results of each type.*
3. *Display a summary of the test results, indicating the number of students who passed and the number who failed.*
4. *If more than eight students passed the exam, print the message "Bonus to instructor!"*

The complete pseudocode appears in Fig. C.7. The Java class that implements the pseudocode algorithm and two sample executions are shown in Fig. C.8. Lines 13–16 of `main` declare the variables that method `processExamResults` of class `Analysis` uses to process the examination results. Several of these declarations use Java's ability to incorporate variable initialization into declarations (`passes` is assigned `0`, `failures 0` and `studentCounter 1`). Looping programs may require initialization at the beginning of each repetition—normally performed by assignment statements rather than in declarations. Java requires that local variables be initialized before their values are used in an expression.

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result

    If the student passed
        Add one to passes
    Else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print "Bonus to instructor!"
```

Fig. C.7 | Pseudocode for examination-results problem.

The `while` statement (lines 19–33) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if...else` statement (lines 26–29) for processing each result is *nested* in the `while` statement. If the `result` is `1`, the `if…else` statement increments `passes`; otherwise, it assumes the `result` is 2 and increments `failures`. Line 32 increments `studentCounter` before the loop condition is tested again at line 19. After 10 values have been input, the loop terminates and line 36 displays the number of `passes` and `failures`. The `if` statement at lines 39–40 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`.

```
 1  // Fig. C.8: Analysis.java
 2  // Analysis of examination results using nested control statements.
 3  import java.util.Scanner; // class uses class Scanner
 4
 5  public class Analysis
 6  {
 7     public static void main( String[] args )
 8     {
 9        // create Scanner to obtain input from command window
10        Scanner input = new Scanner( System.in );
11
12        // initializing variables in declarations
13        int passes = 0; // number of passes
14        int failures = 0; // number of failures
15        int studentCounter = 1; // student counter
16        int result; // one exam result (obtains value from user)
17
18        // process 10 students using counter-controlled loop
19        while ( studentCounter <= 10 )
20        {
21           // prompt user for input and obtain value from user
22           System.out.print( "Enter result (1 = pass, 2 = fail): " );
23           result = input.nextInt();
24
25           // if...else is nested in the while statement
26           if ( result == 1 )          // if result 1,
27              passes = passes + 1;     // increment passes;
28           else                        // else result is not 1, so
29              failures = failures + 1; // increment failures
30
31           // increment studentCounter so loop eventually terminates
32           studentCounter = studentCounter + 1;
33        } // end while
34
35        // termination phase; prepare and display results
36        System.out.printf( "Passed: %d\nFailed: %d\n", passes, failures );
37
38        // determine whether more than 8 students passed
39        if ( passes > 8 )
40           System.out.println( "Bonus to instructor!" );
41     } // end main
42  } // end class Analysis
```

Fig. C.8 | Analysis of examination results using nested control statements. (Part 1 of 2.)

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

Fig. C.8 | Analysis of examination results using nested control statements. (Part 2 of 2.)

During the sample execution, the condition at line 39 of method `main` is `true`—more than eight students passed the exam, so the program outputs a message to bonus the instructor.

This example contains only one class, with method `main` performing all the class's work. Occasionally, when it does not make sense to try to create a *reusable* class to demonstrate a concept, we'll place the program's statements entirely within the `main` method of a single class.

C.11 Compound Assignment Operators

The **compound assignment operators** abbreviate assignment expressions. Statements like

```
    variable = variable  operator  expression;
```

where *operator* is one of the binary operators `+`, `-`, `*`, `/` or `%` (or others we discuss later in the text) can be written in the form

```
    variable  operator=  expression;
```

For example, you can abbreviate the statement

```
    c = c + 3;
```

with the **addition compound assignment operator**, **`+=`**, as

```
    c += 3;
```

The `+=` operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on the left of the operator. Thus, the assignment expression `c += 3` adds `3` to `c`. Figure C.9 shows the arithmetic compound assignment operators, sample expressions using the operators and explanations of what the operators do.

C.12 Increment and Decrement Operators

Java provides two unary operators (summarized in Fig. C.10) for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary **increment operator**, **`++`**, and the unary **decrement operator**, **`--`**. A program can increment by 1 the value of a vari-

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to `c`
`-=`	`d -= 4`	`d = d - 4`	1 to `d`
`*=`	`e *= 5`	`e = e * 5`	20 to `e`
`/=`	`f /= 3`	`f = f / 3`	2 to `f`
`%=`	`g %= 9`	`g = g % 9`	3 to `g`

Fig. C.9 | Arithmetic compound assignment operators.

Operator	Operator name	Sample expression	Explanation
`++`	prefix increment	`++a`	Increment `a` by `1`, then use the new value of `a` in the expression in which `a` resides.
`++`	postfix increment	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by `1`.
`--`	prefix decrement	`--b`	Decrement `b` by `1`, then use the new value of `b` in the expression in which `b` resides.
`--`	postfix decrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by `1`.

Fig. C.10 | Increment and decrement operators.

able called `c` using the increment operator, `++`, rather than the expression `c = c + 1` or `c += 1`. An increment or decrement operator that's prefixed to (placed before) a variable is referred to as the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator that's postfixed to (placed after) a variable is referred to as the **postfix increment** or **postfix decrement operator**, respectively.

Using the prefix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable is known as **preincrementing** (or **predecrementing**). This causes the variable to be incremented (decremented) by 1; then the new value of the variable is used in the expression in which it appears. Using the postfix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable is known as **postincrementing** (or **postdecrementing**). This causes the current value of the variable to be used in the expression in which it appears; then the variable's value is incremented (decremented) by 1.

Figure C.11 demonstrates the difference between the prefix increment and postfix increment versions of the `++` increment operator. The decrement operator (`--`) works similarly. Line 11 initializes the variable `c` to `5`, and line 12 outputs `c`'s initial value. Line 13 outputs the value of the expression `c++`. This expression postincrements the variable `c`, so `c`'s original value (`5`) is output, then `c`'s value is incremented (to 6). Thus, line 13 outputs `c`'s initial value (`5`) again. Line 14 outputs `c`'s new value (`6`) to prove that the variable's value was indeed incremented in line 13.

```
 1  // Fig. C.11: Increment.java
 2  // Prefix increment and postfix increment operators.
 3
 4  public class Increment
 5  {
 6     public static void main( String[] args )
 7     {
 8        int c;
 9
10        // demonstrate postfix increment operator
11        c = 5; // assign 5 to c
12        System.out.println( c );   // prints 5
13        System.out.println( c++ ); // prints 5 then postincrements
14        System.out.println( c );   // prints 6
15
16        System.out.println(); // skip a line
17
18        // demonstrate prefix increment operator
19        c = 5; // assign 5 to c
20        System.out.println( c );   // prints 5
21        System.out.println( ++c ); // preincrements then prints 6
22        System.out.println( c );   // prints 6
23     } // end main
24  } // end class Increment
```

```
5
5
6

5
6
6
```

Fig. C.11 | Preincrementing and postincrementing.

Line 19 resets c's value to 5, and line 20 outputs c's value. Line 21 outputs the value of the expression ++c. This expression preincrements c, so its value is incremented; then the new value (6) is output. Line 22 outputs c's value again to show that the value of c is still 6 after line 21 executes.

When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the same effect, and the prefix decrement and postfix decrement forms have the same effect. It's only when a variable appears in the context of a larger expression that preincrementing and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

C.13 Primitive Types

The table in Appendix L lists the eight primitive types in Java. Like its predecessor languages C and C++, Java requires all variables to have a type. For this reason, Java is referred to as a **strongly typed language**.

In C and C++, programmers frequently have to write separate versions of programs to support different computer platforms, because the primitive types are not guaranteed to be identical from computer to computer. For example, an `int` value on one machine might be represented by 16 bits (2 bytes) of memory, on a second machine by 32 bits (4 bytes) of memory, and on another machine by 64 bits (8 bytes) of memory. In Java, `int` values are always 32 bits (4 bytes).

Portability Tip C.1
The primitive types in Java are portable across all computer platforms that support Java.

Each type in Appendix L is listed with its size in bits (there are eight bits to a byte) and its range of values. Because the designers of Java want to ensure portability, they use internationally recognized standards for character formats (Unicode; for more information, visit `www.unicode.org`) and floating-point numbers (IEEE 754; for more information, visit `grouper.ieee.org/groups/754/`).

C.14 Essentials of Counter-Controlled Repetition

This section uses the `while` repetition statement introduced in Section C.7 to formalize the elements required to perform counter-controlled repetition, which requires

1. a **control variable** (or loop counter)
2. the **initial value** of the control variable
3. the **increment** (or **decrement**) by which the control variable is modified each time through the loop (also known as **each iteration of the loop**)
4. the **loop-continuation condition** that determines if looping should continue.

To see these elements of counter-controlled repetition, consider the application of Fig. C.12, which uses a loop to display the numbers from 1 through 10.

```java
// Fig. C.12: WhileCounter.java
// Counter-controlled repetition with the while repetition statement.

public class WhileCounter
{
   public static void main( String[] args )
   {
      int counter = 1; // declare and initialize control variable

      while ( counter <= 10 ) // loop-continuation condition
      {
         System.out.printf( "%d  ", counter );
         ++counter; // increment control variable by 1
      } // end while

      System.out.println(); // output a newline
   } // end main
} // end class WhileCounter
```

Fig. C.12 | Counter-controlled repetition with the `while` repetition statement. (Part 1 of 2.)

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

Fig. C.12 | Counter-controlled repetition with the `while` repetition statement. (Part 2 of 2.)

In Fig. C.12, the elements of counter-controlled repetition are defined in lines 8, 10 and 13. Line 8 declares the control variable (`counter`) as an `int`, reserves space for it in memory and sets its initial value to `1`. Line 12 displays control variable `counter`'s value during each iteration of the loop. Line 13 increments the control variable by 1 for each iteration of the loop. The loop-continuation condition in the `while` (line 10) tests whether the value of the control variable is less than or equal to `10` (the final value for which the condition is `true`). The program performs the body of this `while` even when the control variable is `10`. The loop terminates when the control variable exceeds `10` (i.e., `counter` becomes `11`).

C.15 for Repetition Statement

Java also provides the **`for` repetition statement**, which specifies the counter-controlled-repetition details in a single line of code. Figure C.13 reimplements the application of Fig. C.12 using `for`.

```
 1  // Fig. C.13: ForCounter.java
 2  // Counter-controlled repetition with the for repetition statement.
 3
 4  public class ForCounter
 5  {
 6     public static void main( String[] args )
 7     {
 8        // for statement header includes initialization,
 9        // loop-continuation condition and increment
10        for ( int counter = 1; counter <= 10; ++counter )
11           System.out.printf( "%d  ", counter );
12
13        System.out.println(); // output a newline
14     } // end main
15  } // end class ForCounter
```

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

Fig. C.13 | Counter-controlled repetition with the `for` repetition statement.

When the `for` statement (lines 10–11) begins executing, the control variable `counter` is declared and initialized to `1`. Next, the program checks the loop-continuation condition, `counter <= 10`, which is between the two required semicolons. Because the initial value of `counter` is `1`, the condition initially is true. Therefore, the body statement (line 11) displays control variable `counter`'s value, namely `1`. After executing the loop's body, the program increments `counter` in the expression `++counter`, which appears to the right of the second semicolon. Then the loop-continuation test is performed again to determine whether the program should continue with the next iteration of the loop. At this point, the control variable's value is `2`, so the condition is still true (the final value is not exceeded)—thus, the program performs the body statement again (i.e., the next iteration of the loop). This process continues until the numbers 1 through 10 have been displayed and the `counter`'s value becomes `11`, causing the loop-continuation test to fail and repetition to terminate (after 10 repetitions of the loop body). Then the program performs the first statement after the `for`—in this case, line 13.

Figure C.13 uses (in line 10) the loop-continuation condition `counter <= 10`. If you incorrectly specified `counter < 10` as the condition, the loop would iterate only nine times. This is a common logic error called an **off-by-one error**.

A Closer Look at the `for` Statement's Header

Figure C.14 takes a closer look at the `for` statement in Fig. C.13. The `for`'s first line (including the keyword `for` and everything in parentheses after `for`)—line 10 in Fig. C.13—is sometimes called the **`for` statement header**. The `for` header "does it all"—it specifies each item needed for counter-controlled repetition with a control variable. If there's more than one statement in the body of the `for`, braces are required to define the body of the loop. If the loop-continuation condition is initially `false`, the program does not execute the `for` statement's body—execution proceeds with the statement following the `for`.

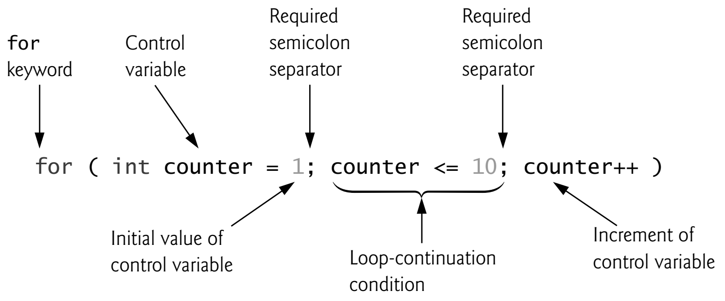

for keyword
Control variable
Required semicolon separator
Required semicolon separator
for (int counter = 1; counter <= 10; counter++)
Initial value of control variable
Loop-continuation condition
Increment of control variable

Fig. C.14 | `for` statement header components.

Scope of a `for` Statement's Control Variable

If the *initialization* expression in the `for` header declares the control variable (i.e., the control variable's type is specified before the variable name, as in Fig. C.13), the control variable can be used *only* in that `for` statement—it will not exist outside it. This restricted use is known as the variable's **scope**. The scope of a variable defines where it can be used in a program. For example, a local variable can be used *only* in the method that declares it and *only* from the point of declaration through the end of the method.

Expressions in a `for` Statement's Header Are Optional

All three expressions in a `for` header are optional. If the *loopContinuationCondition* is omitted, Java assumes that the loop-continuation condition is always *true*, thus creating an infinite loop. You might omit the *initialization* expression if the program initializes the control variable before the loop. You might omit the *increment* expression if the program calculates the increment with statements in the loop's body or if no increment is needed. The increment expression in a `for` acts as if it were a standalone statement at the end of the `for`'s body.

C.16 Examples Using the for Statement

The following examples show techniques for varying the control variable in a `for` statement. In each case, we write the appropriate `for` header. Note the change in the relational operator for loops that *decrement* the control variable to count downward.

a) Vary the control variable from `1` to `100` in *increments* of `1`.

```
for ( int i = 1; i <= 100; ++i )
```

b) Vary the control variable from `100` to `1` in *decrements* of `1`.

```
for ( int i = 100; i >= 1; --i )
```

c) Vary the control variable from `7` to `77` in *increments* of `7`.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Vary the control variable from `20` to `2` in *decrements* of `2`.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Vary the control variable over the values `2, 5, 8, 11, 14, 17, 20`.

```
for ( int i = 2; i <= 20; i += 3 )
```

f) Vary the control variable over the values `99, 88, 77, 66, 55, 44, 33, 22, 11, 0`.

```
for ( int i = 99; i >= 0; i -= 11 )
```

Application: Compound-Interest Calculations

Let's use the `for` statement to compute compound interest. Consider the following problem:

A person invests $1000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p\,(1 + r)^n$$

where

p is the original amount invested (i.e., the principal)
r is the annual interest rate (e.g., use 0.05 for 5%)
n is the number of years
a is the amount on deposit at the end of the nth year.

The solution to this problem (Fig. C.15) involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. Lines 8–10 in method `main` declare `double` variables `amount`, `principal` and `rate`, and initialize `principal` to `1000.0` and `rate` to `0.05`. Java treats floating-point constants like `1000.0` and `0.05` as type `double`. Similarly, Java treats whole-number constants like `7` and `-22` as type `int`.

```
1  // Fig. C.15: Interest.java
2  // Compound-interest calculations with for.
3
```

Fig. C.15 | Compound-interest calculations with `for`. (Part 1 of 2.)

```
 4  public class Interest
 5  {
 6     public static void main( String[] args )
 7     {
 8        double amount; // amount on deposit at end of each year
 9        double principal = 1000.0; // initial amount before interest
10        double rate = 0.05; // interest rate
11
12        // display headers
13        System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );
14
15        // calculate amount on deposit for each of ten years
16        for ( int year = 1; year <= 10; ++year )
17        {
18           // calculate new amount for specified year
19           amount = principal * Math.pow( 1.0 + rate, year );
20
21           // display the year and the amount
22           System.out.printf( "%4d%,20.2f\n", year, amount );
23        } // end for
24     } // end main
25  } // end class Interest
```

```
Year   Amount on deposit
   1            1,050.00
   2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
   9            1,551.33
  10            1,628.89
```

Fig. C.15 | Compound-interest calculations with for. (Part 2 of 2.)

Formatting Strings with Field Widths and Justification
Line 13 outputs two column headers. The first column displays the year and the second the amount on deposit at the end of that year. We use the format specifier %20s to output the String "Amount on Deposit". The integer 20 between the % and the conversion character s indicates that the value should be displayed in a **field width** of 20—that is, printf displays the value with at least 20 character positions. If the value requires fewer than 20 character positions (17 in this example), the value is **right justified** in the field by default. If the year value to be output were more than four character positions wide, the field width would be extended to the right to accommodate the entire value—this would push the amount field to the right, upsetting the neat columns of our tabular output. To output values **left justified**, simply precede the field width with the **minus sign (–) formatting flag** (e.g., %-20s).

Performing the Interest Calculations
The for statement (lines 16–23) executes its body 10 times, varying control variable year from 1 to 10 in increments of 1. This loop terminates when year becomes 11. (Variable year represents *n* in the problem statement.)

Classes provide methods that perform common tasks on objects. In fact, most methods must be called on a specific object. For example, to output text in Fig. C.15, line 13 calls method `printf` on the `System.out` object. Many classes also provide methods that perform common tasks and do *not* require objects. These are called `static` methods. For example, Java does not include an exponentiation operator, so the designers of Java's `Math` class defined `static` method `pow` for raising a value to a power. You can call a `static` method by specifying the class name followed by a dot (`.`) and the method name, as in

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

In Appendix D, you'll learn how to implement `static` methods in your own classes.

We use `static` method **`pow`** of class **`Math`** to perform the compound-interest calculation in Fig. C.15. `Math.pow(`*x*`,` *y*`)` calculates the value of *x* raised to the y^{th} power. The method receives two `double` arguments and returns a `double` value. Line 19 performs the calculation $a = p(1 + r)^n$, where *a* is `amount`, *p* is `principal`, *r* is `rate` and *n* is `year`. Class `Math` is defined in package `java.lang`, so you do *not* need to import class `Math` to use it.

Formatting Floating-Point Numbers

After each calculation, line 22 outputs the year and the amount on deposit at the end of that year. The year is output in a field width of four characters (as specified by `%4d`). The amount is output as a floating-point number with the format specifier `%,20.2f`. The **comma (,) formatting flag** indicates that the floating-point value should be output with a **grouping separator**. The actual separator used is specific to the user's locale (i.e., country). For example, in the United States, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in 1,234.45. The number `20` in the format specification indicates that the value should be output right justified in a field width of 20 characters. The `.2` specifies the formatted number's precision—in this case, the number is rounded to the nearest hundredth and output with two digits to the right of the decimal point.

C.17 do…while Repetition Statement

The **`do…while` repetition statement** is similar to the `while` statement. In the `while`, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body; if the condition is false, the body *never* executes. The `do…while` statement tests the loop-continuation condition *after* executing the loop's body; therefore, *the body always executes at least once*. When a `do…while` statement terminates, execution continues with the next statement in sequence. Figure C.16 uses a `do…while` (lines 10–14) to output the numbers 1–10.

```
1  // Fig. C.16: DoWhileTest.java
2  // do...while repetition statement.
3
4  public class DoWhileTest
5  {
6     public static void main( String[] args )
7     {
```

Fig. C.16 | `do…while` repetition statement. (Part 1 of 2.)

```
 8        int counter = 1; // initialize counter
 9
10        do
11        {
12           System.out.printf( "%d  ", counter );
13           ++counter;
14        } while ( counter <= 10 ); // end do...while
15
16        System.out.println(); // outputs a newline
17     } // end main
18  } // end class DoWhileTest
```

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

Fig. C.16 | `do...while` repetition statement. (Part 2 of 2.)

Line 8 declares and initializes control variable `counter`. Upon entering the `do...while` statement, line 12 outputs `counter`'s value and line 13 increments `counter`. Then the program evaluates the loop-continuation test at the *bottom* of the loop (line 14). If the condition is true, the loop continues from the first body statement (line 12). If the condition is false, the loop terminates and the program continues with the next statement after the loop.

C.18 `switch` Multiple-Selection Statement

Sections C.5–C.6 discussed the `if` single-selection and the `if...else` double-selection statements. The **`switch` multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type `byte`, `short`, `int` or `char`.

`GradeBook` Class with `switch` Statement to Count A, B, C, D and F Grades

Figure C.17 enhances the `GradeBook` case study that we began presenting in Appendix B. The new version we now present not only calculates the average of a set of numeric grades entered by the user, but uses a `switch` statement to determine whether each grade is the equivalent of an A, B, C, D or F and to increment the appropriate grade counter. The class also displays a summary of the number of students who received each grade. Refer to Fig. C.18 for sample inputs and outputs of the `GradeBookTest` application that uses class `GradeBook` to process a set of grades.

```
// Fig. C.17: GradeBook.java
// GradeBook class uses the switch statement to count letter grades.
import java.util.Scanner; // program uses class Scanner

public class GradeBook
{
   private String courseName; // name of course this GradeBook represents
   // int instance variables are initialized to 0 by default
   private int total; // sum of grades
   private int gradeCounter; // number of grades entered
```

Fig. C.17 | `GradeBook` class uses the `switch` statement to count letter grades. (Part 1 of 4.)

```
   private int aCount; // count of A grades
   private int bCount; // count of B grades
   private int cCount; // count of C grades
   private int dCount; // count of D grades
   private int fCount; // count of F grades

   // constructor initializes courseName;
   public GradeBook( String name )
   {
      courseName = name; // initializes courseName
   } // end constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // input arbitrary number of grades from user
   public void inputGrades()
   {
      Scanner input = new Scanner( System.in );

      int grade; // grade entered by user

      System.out.printf( "%s\n%s\n   %s\n   %s\n",
         "Enter the integer grades in the range 0-100.",
         "Type the end-of-file indicator to terminate input:",
         "On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter",
         "On Windows type <Ctrl> z then press Enter" );

      // loop until user enters the end-of-file indicator
      while ( input.hasNext() )
      {
         grade = input.nextInt(); // read grade
         total += grade; // add grade to total
         ++gradeCounter; // increment number of grades

```

Fig. C.17 | GradeBook class uses the switch statement to count letter grades. (Part 2 of 4.)

```
            // call method to increment appropriate counter
            incrementLetterGradeCounter( grade );
         } // end while
      } // end method inputGrades

      // add 1 to appropriate counter for specified grade
      private void incrementLetterGradeCounter( int grade )
      {
         // determine which grade was entered
         switch ( grade / 10 )
         {
            case 9:  // grade was between 90
            case 10: // and 100, inclusive
               ++aCount; // increment aCount
               break; // necessary to exit switch

            case 8: // grade was between 80 and 89
               ++bCount; // increment bCount
               break; // exit switch

            case 7: // grade was between 70 and 79
               ++cCount; // increment cCount
               break; // exit switch

            case 6: // grade was between 60 and 69
               ++dCount; // increment dCount
               break; // exit switch

            default: // grade was less than 60
               ++fCount; // increment fCount
               break; // optional; will exit switch anyway
         } // end switch
      } // end method incrementLetterGradeCounter

      // display a report based on the grades entered by the user
      public void displayGradeReport()
      {
         System.out.println( "\nGrade Report:" );

         // if user entered at least one grade...
         if ( gradeCounter != 0 )
         {
            // calculate average of all grades entered
            double average = (double) total / gradeCounter;

            // output summary of results
            System.out.printf( "Total of the %d grades entered is %d\n",
               gradeCounter, total );
            System.out.printf( "Class average is %.2f\n", average );
            System.out.printf( "%s\n%s%d\n%s%d\n%s%d\n%s%d\n%s%d\n",
               "Number of students who received each grade:",
               "A: ", aCount,   // display number of A grades
               "B: ", bCount,   // display number of B grades
```

Fig. C.17 | GradeBook class uses the switch statement to count letter grades. (Part 3 of 4.)

```
116             "C: ", cCount,   // display number of C grades
117             "D: ", dCount,   // display number of D grades
118             "F: ", fCount ); // display number of F grades
119       } // end if
120       else // no grades were entered, so output appropriate message
121          System.out.println( "No grades were entered" );
122    } // end method displayGradeReport
123 } // end class GradeBook
```

Fig. C.17 | `GradeBook` class uses the `switch` statement to count letter grades. (Part 4 of 4.)

Like earlier versions of the class, class `GradeBook` (Fig. C.17) declares instance variable `courseName` (line 7) and contains methods `setCourseName` (lines 24–27), `getCourseName` (lines 30–33) and `displayMessage` (lines 36–41), which set the course name, store the course name and display a welcome message to the user, respectively. The class also contains a constructor (lines 18–21) that initializes the course name.

Class `GradeBook` also declares instance variables `total` (line 9) and `gradeCounter` (line 10), which keep track of the sum of the grades entered by the user and the number of grades entered, respectively. Lines 11–15 declare counter variables for each grade category. Class `GradeBook` maintains `total`, `gradeCounter` and the five letter-grade counters as instance variables so that they can be used or modified in any of the class's methods. The class's constructor (lines 18–21) sets only the course name, because the remaining seven instance variables are `int`s and are initialized to `0` by default.

Class `GradeBook` contains three additional methods—`inputGrades`, `incrementLetterGradeCounter` and `displayGradeReport`. Method `inputGrades` (lines 44–66) reads an arbitrary number of integer grades from the user using sentinel-controlled repetition and updates instance variables `total` and `gradeCounter`. This method calls method `incrementLetterGradeCounter` (lines 69–95) to update the appropriate letter-grade counter for each grade entered. Method `displayGradeReport` (lines 98–122) outputs a report containing the total of all grades entered, the average of the grades and the number of students who received each letter grade. Let's examine these methods in more detail.

Method `inputGrades`

Line 48 in method `inputGrades` declares variable `grade`, which will store the user's input. Lines 50–54 prompt the user to enter integer grades and to type the end-of-file indicator to terminate the input. The **end-of-file indicator** is a system-dependent keystroke combination which the user enters to indicate that there's no more data to input.

On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence

<Ctrl> d

on a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key. On Windows systems, end-of-file can be entered by typing

<Ctrl> z

[*Note:* On some systems, you must press *Enter* after typing the end-of-file key sequence. Also, Windows typically displays the characters `^Z` on the screen when the end-of-file indicator is typed, as shown in the output of Fig. C.18.]

The while statement (lines 57–65) obtains the user input. The condition at line 57 calls Scanner method **hasNext** to determine whether there's more data to input. This method returns the boolean value true if there's more data; otherwise, it returns false. The returned value is then used as the value of the condition in the while statement. Method hasNext returns false once the user types the end-of-file indicator.

Line 59 inputs a grade value from the user. Line 60 adds grade to total. Line 61 increments gradeCounter. The class's displayGradeReport method uses these variables to compute the average of the grades. Line 64 calls the class's incrementLetterGrade-Counter method (declared in lines 69–95) to increment the appropriate letter-grade counter based on the numeric grade entered.

Method ***incrementLetterGradeCounter***

Method incrementLetterGradeCounter contains a switch statement (lines 72–94) that determines which counter to increment. We assume that the user enters a valid grade in the range 0–100. A grade in the range 90–100 represents A, 80–89 represents B, 70–79 represents C, 60–69 represents D and 0–59 represents F. The switch statement consists of a block that contains a sequence of **case labels** and an optional **default case**. These are used in this example to determine which counter to increment based on the grade.

When the flow of control reaches the switch, the program evaluates the expression in the parentheses (grade / 10) following keyword switch. This is the switch's **controlling expression**. The program compares this expression's value (which must evaluate to an integral value of type byte, char, short or int) with each case label. The controlling expression in line 72 performs integer division, which *truncates the fractional part* of the result. Thus, when we divide a value from 0 to 100 by 10, the result is always a value from 0 to 10. We use several of these values in our case labels. For example, if the user enters the integer 85, the controlling expression evaluates to 8. The switch compares 8 with each case label. If a match occurs (case 8: at line 79), the program executes that case's statements. For the integer 8, line 80 increments bCount, because a grade in the 80s is a B. The **break statement** (line 81) causes program control to proceed with the first statement after the switch—in this program, we reach the end of method incrementLetterGrade-Counter's body, so the method terminates and control returns to line 65 in method inputGrades (the first line after the call to incrementLetterGradeCounter). Line 65 is the end of a while loop's body, so control flows to the while's condition (line 57) to determine whether the loop should continue executing.

The cases in our switch explicitly test for the values 10, 9, 8, 7 and 6. Note the cases at lines 74–75 that test for the values 9 and 10 (both of which represent the grade A). Listing cases consecutively in this manner with no statements between them enables the cases to perform the same set of statements—when the controlling expression evaluates to 9 or 10, the statements in lines 76–77 will execute. The switch statement does not provide a mechanism for testing ranges of values, so every value you need to test must be listed in a separate case label. Each case can have multiple statements. The switch statement differs from other control statements in that it does *not* require braces around multiple statements in a case.

Without break statements, each time a match occurs in the switch, the statements for that case and subsequent cases execute until a break statement or the end of the switch is encountered. (This feature is helpful for writing a concise program that displays the iterative song "The Twelve Days of Christmas").

If no match occurs between the controlling expression's value and a `case` label, the `default` case (lines 91–93) executes. We use the `default` case in this example to process all controlling-expression values that are less than `6`—that is, all failing grades. If no match occurs and the `switch` does not contain a `default` case, program control simply continues with the first statement after the `switch`.

GradeBookTest* Class That Demonstrates Class *GradeBook

Class `GradeBookTest` (Fig. C.18) creates a `GradeBook` object (lines 10–11). Line 13 invokes the object's `displayMessage` method to output a welcome message to the user. Line 14 invokes the object's `inputGrades` method to read a set of grades from the user and keep track of the sum of all the grades entered and the number of grades. Recall that method `inputGrades` also calls method `incrementLetterGradeCounter` to keep track of the number of students who received each letter grade. Line 15 invokes method `displayGradeReport` of class `GradeBook`, which outputs a report based on the grades entered (as in the input/output window in Fig. C.18). Line 103 of class `GradeBook` (Fig. C.17) determines whether the user entered at least one grade—this helps us avoid dividing by zero. If so, line 106 calculates the average of the grades. Lines 109–118 then output the total of all the grades, the class average and the number of students who received each letter grade. If no grades were entered, line 121 outputs an appropriate message. The output in Fig. C.18 shows a sample grade report based on 10 grades.

```java
 1  // Fig. C.18: GradeBookTest.java
 2  // Create GradeBook object, input grades and display grade report.
 3
 4  public class GradeBookTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // create GradeBook object myGradeBook and
 9        // pass course name to constructor
10        GradeBook myGradeBook = new GradeBook(
11           "CS101 Introduction to Java Programming" );
12
13        myGradeBook.displayMessage(); // display welcome message
14        myGradeBook.inputGrades(); // read grades from user
15        myGradeBook.displayGradeReport(); // display report based on grades
16     } // end main
17  } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
```

Fig. C.18 | Create `GradeBook` object, input grades and display grade report. (Part 1 of 2.)

```
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80


Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. C.18 | Create `GradeBook` object, input grades and display grade report. (Part 2 of 2.)

Class `GradeBookTest` (Fig. C.18) does not directly call `GradeBook` method `incrementLetterGradeCounter` (lines 69–95 of Fig. C.17). This method is used exclusively by method `inputGrades` of class `GradeBook` to update the appropriate letter-grade counter as each new grade is entered by the user. Method `incrementLetterGradeCounter` exists solely to support the operations of `GradeBook`'s other methods, so it's declared `private`.

The `break` statement is not required for the `switch`'s last `case` (or the optional `default` case, when it appears last), because execution continues with the next statement after the `switch`.

Notes on the Expression in Each `case` of a `switch`

When using the `switch` statement, remember that each `case` must contain a constant integral expression—that is, any combination of integer constants that evaluates to a constant integer value (e.g., –7, 0 or 221). An integer constant is simply an integer value. In addition, you can use **character constants**—specific characters in single quotes, such as `'A'`, `'7'` or `'$'`—which represent the integer values of characters and `enum` constants (introduced in Section D.10).

The expression in each `case` can also be a **constant variable**—a variable containing a value which does not change for the entire program. Such a variable is declared with keyword `final` (discussed in Appendix D). Java has a feature called *enumerations*, which we also present in Appendix D. Enumeration constants can also be used in `case` labels.

Using `String`s in `switch` Statements (New in Java SE 7)

As of Java SE 7, you can use `String`s in a `switch` statement's controlling expression and in `case` labels. For example, you might want to use a city's name to obtain the corresponding ZIP code. Assuming that `city` and `zipCode` are `String` variables, the following `switch` statement performs this task for three cities:

```
      switch( city )
      {
         case "Maynard":
            zipCode = "01754";
            break;
         case "Marlborough":
            zipCode = "01752";
            break;
         case "Framingham":
            zipCode = "01701";
            break;
      } // end switch
```

C.19 break and continue Statements

In addition to selection and repetition statements, Java provides statements `break` and **continue** to alter the flow of control. The preceding section showed how `break` can be used to terminate a `switch` statement's execution. This section discusses how to use `break` in repetition statements.

break Statement

The `break` statement, when executed in a `while`, `for`, `do…while` or `switch`, causes immediate exit from that statement. Execution continues with the first statement after the control statement. Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch`.

continue Statement

The `continue` statement, when executed in a `while`, `for` or `do…while`, skips the remaining statements in the loop body and proceeds with the *next iteration* of the loop. In `while` and `do…while` statements, the program evaluates the loop-continuation test immediately after the `continue` statement executes. In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

C.20 Logical Operators

Java's **logical operators** enable you to form more complex conditions by *combining* simple conditions. The logical operators are `&&` (conditional AND), `||` (conditional OR), `&` (boolean logical AND), `|` (boolean logical inclusive OR), `^` (boolean logical exclusive OR) and `!` (logical NOT). [*Note:* The `&`, `|` and `^` operators are also bitwise operators when they're applied to integral operands.]

Conditional AND (&&) Operator

Suppose we wish to ensure at some point in a program that two conditions are *both* true before we choose a certain path of execution. In this case, we can use the **&&** (**conditional AND**) operator, as follows:

```
      if ( gender == FEMALE && age >= 65 )
         ++seniorFemales;
```

This `if` statement contains two simple conditions. The condition `gender == FEMALE` compares variable `gender` to the constant `FEMALE` to determine whether a person is female. The condition age >= 65 might be evaluated to determine whether a person is a senior citizen. The if statement considers the combined condition

```
gender == FEMALE && age >= 65
```

which is true if and only if *both* simple conditions are true. In this case, the if statement's body increments seniorFemales by 1. If either or both of the simple conditions are false, the program skips the increment. Some programmers find that the preceding combined condition is more readable when redundant parentheses are added, as in:

```
( gender == FEMALE ) && ( age >= 65 )
```

The table in Fig. C.19 summarizes the && operator. The table shows all four possible combinations of false and true values for *expression1* and *expression2*. Such tables are called **truth tables**. Java evaluates to false or true all expressions that include relational operators, equality operators or logical operators.

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. C.19 | && (conditional AND) operator truth table.

Conditional OR (||) Operator

Now suppose we wish to ensure that *either or both* of two conditions are true before we choose a certain path of execution. In this case, we use the || (**conditional OR**) operator, as in the following program segment:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   System.out.println ( "Student grade is A" );
```

This statement also contains two simple conditions. The condition semesterAverage >= 90 evaluates to determine whether the student deserves an A in the course because of a solid performance throughout the semester. The condition finalExam >= 90 evaluates to determine whether the student deserves an A in the course because of an outstanding performance on the final exam. The if statement then considers the combined condition

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

and awards the student an A if *either or both* of the simple conditions are true. The only time the message "Student grade is A" is *not* printed is when *both* of the simple conditions are *false*. Figure C.20 is a truth table for operator conditional OR (||). Operator && has a higher precedence than operator ||. Both operators associate from left to right.

Short-Circuit Evaluation of Complex Conditions

The parts of an expression containing && or || operators are evaluated *only* until it's known whether the condition is true or false. Thus, evaluation of the expression

```
( gender == FEMALE ) && ( age >= 65 )
```

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. C.20 | `||` (conditional OR) operator truth table.

stops immediately if `gender` is not equal to `FEMALE` (i.e., the entire expression is `false`) and continues if `gender` *is* equal to `FEMALE` (i.e., the entire expression could still be `true` if the condition `age >= 65` is `true`). This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation**.

Boolean Logical AND (&) and Boolean Logical Inclusive OR (|) Operators

The **boolean logical AND** (`&`) and **boolean logical inclusive OR** (`|`) operators are identical to the `&&` and `||` operators, except that the `&` and `|` operators *always* evaluate *both* of their operands (i.e., they do *not* perform short-circuit evaluation). So, the expression

```
      ( gender == 1 ) & ( age >= 65 )
```

evaluates `age >= 65` *regardless* of whether `gender` is equal to `1`. This is useful if the right operand of the boolean logical AND or boolean logical inclusive OR operator has a required **side effect**—a modification of a variable's value. For example, the expression

```
      ( birthday == true ) | ( ++age >= 65 )
```

guarantees that the condition `++age >= 65` will be evaluated. Thus, the variable `age` is incremented, regardless of whether the overall expression is `true` or `false`.

Error-Prevention Tip C.1
For clarity, avoid expressions with side effects in conditions. The side effects may seem clever, but they can make it harder to understand code and can lead to subtle logic errors.

Boolean Logical Exclusive OR (^)

A simple condition containing the **boolean logical exclusive OR** (`^`) operator is `true` *if and only if one of its operands is* `true` *and the other is* `false`. If both are `true` or both are `false`, the entire condition is `false`. Figure C.21 is a truth table for the boolean logical exclusive OR operator (`^`). This operator is guaranteed to evaluate *both* of its operands.

expression1	expression2	expression1 ^ expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`false`

Fig. C.21 | `^` (boolean logical exclusive OR) operator truth table.

Logical Negation (`!`*) Operator*

The `!` (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition. Unlike the logical operators `&&`, `||`, `&`, `|` and `^`, which are ***binary*** operators that combine two conditions, the logical negation operator is a *unary* operator that has only a single condition as an operand. The operator is placed *before* a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`, as in the program segment

```
if ( ! ( grade == sentinelValue ) )
   System.out.printf( "The next grade is %d\n", grade );
```

which executes the `printf` call only if `grade` is *not* equal to `sentinelValue`. The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement may also be written as follows:

```
if ( grade != sentinelValue )
   System.out.printf( "The next grade is %d\n", grade );
```

This flexibility can help you express a condition in a more convenient manner. Figure C.22 is a truth table for the logical negation operator.

expression	!expression
`false`	`true`
`true`	`false`

Fig. C.22 | `!` (logical negation, or logical NOT) operator truth table.

C.21 Wrap-Up

This appendix presented basic problem solving for building classes and developing methods for these classes. We demonstrated how to construct an algorithm (i.e., an approach to solving a problem), then how to refine the algorithm through several phases of pseudocode development, resulting in Java code that can be executed as part of a method. The appendix showed how to use top-down, stepwise refinement to plan out the specific actions that a method must perform and the order in which the method must perform these actions.

Only three types of control structures—sequence, selection and repetition—are needed to develop any problem-solving algorithm. Specifically, this appendix demonstrated the `if` single-selection statement, the `if...else` double-selection statement and the `while` repetition statement. These are some of the building blocks used to construct solutions to many problems. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled repetition, and we used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced Java's compound assignment operators and its increment and decrement operators. We discussed Java's primitive types.

We demonstrated the `for`, `do…while` and `switch` statements. We showed that any algorithm can be developed using combinations of the sequence structure (i.e., statements listed in the order in which they should execute), the three types of selection statements—`if`, `if…else` and `switch`—and the three types of repetition statements—`while`, `do…while` and `for`. We discussed how you can combine these building blocks to utilize proven program-construction and problem-solving techniques. We also introduced Java's logical operators, which enable you to use more complex conditional expressions in control statements. In Appendix D, we examine methods in greater depth.

Self-Review Exercises (Sections C.1–C.13)

C.1 Fill in the blanks in each of the following statements:

a) ________ is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.

b) The process of executing the statements in a program one after the other in the order in which they're written is called ________.

c) When it's not known in advance how many times a set of statements will be repeated, a(n) ________ value can be used to terminate the repetition.

d) Java is a(n) ________ language; it requires all variables to have a type.

e) ________ enable programs to perform statements repeatedly as long as a condition remains true.

C.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `if` statement is a double-selection statement.

b) Java provides the ternary operator (?:), which can be used in place of a `do…while` statement.

c) A nested control statement appears in the body of another control statement.

d) Specifying the order in which statements execute in a program is called program control.

e) A nonfatal logic error causes a program to fail and terminate prematurely.

C.3 Write Java statements to accomplish each of the following tasks:

a) Use one statement to assign the product of `x` and `y` to `z`, then decrement `y` by `1`.

b) Test whether variable `count` is equal to `10`. If it is, print "Count is equal to `10`".

c) Use the ternary operator to check if variable `x` is lesser than `20`, and add `20` to `x` if it is true; if not, subtract `x` by `20` and assign the result to variable `y`.

d) Use one statement to decrement the variable `x` by `5`, then add it to the variable `total` and store the result in variable `total`.

C.4 Write a Java statement to accomplish each of the following tasks:

a) Declare variables `sum` and `x` to be of type `int`.

b) Assign `1` to variable `x`.

c) Assign `0` to variable `sum`.

d) Add variable `x` to variable `sum`, and assign the result to variable `sum`.

e) Print `"The sum is: "`, followed by the value of variable `sum`.

C.5 Determine the value of the variables in the statement `product *= x++;` after the calculation is performed. Assume that all variables are type `int` and initially have the value 5.

C.6 Identify and correct the errors in each of the following sets of code:

a)
```
while ( c <= 5 )
{
   product *= c;
   ++c;
```

b)
```
if ( gender == 1 )
   System.out.println( "Woman" );
else;
   System.out.println( "Man" );
```

C.7 What is wrong with the following `while` statement?

```
while ( z >= 0 )
   sum += z;
```

Self-Review Exercises (Sections C.14–C.20)

C.8 Fill in the blanks in each of the following statements:

a) Typically, ________ statements are used for counter-controlled repetition and ________ statements for sentinel-controlled repetition.
b) The `do...while` statement tests the loop-continuation condition ________ executing the loop's body; therefore, the body always executes at least once.
c) The ________ statement selects among multiple actions based on the possible values of an integer variable or expression.
d) The ________ operator can be used to ensure that two conditions are *both* true before choosing a certain path of execution.
e) If the loop-continuation condition in a `for` header is initially ________, the program does not execute the `for` statement's body.

C.9 State whether each of the following is *true* or *false*. If *false*, explain why.

a) The `default` case is required in the `switch` selection statement.
b) The `break` statement is required in the last case of a `switch` selection statement.
c) The expression `( ( x > y ) && ( a < b ) )` is true if either `x > y` is true or `a < b` is true.
d) An expression containing the `||` operator is true if either or both of its operands are true.
e) Listing cases consecutively with no statements between them enables the cases to perform the same set of statements.

C.10 Write a Java statement or a set of Java statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99, using a `for` statement. Assume that the integer variables `sum` and `count` have been declared.
b) Calculate the value of `2.5` raised to the power of `3`, using the `pow` method.
c) Print the integers from 1 to 20, using a `while` loop and the counter variable `i`. Assume that the variable `i` has been declared, but not initialized. Print only five integers per line. [*Hint:* Use the calculation `i % 5`. When the value of this expression is 0, print a newline character; otherwise, print a tab character. Assume that this code is an application. Use the `System.out.println()` method to output the newline character, and use the `System.out.print( '\t' )` method to output the tab character.]
d) Repeat part (c), using a `for` statement.

C.11 Find the error in each of the following code segments, and explain how to correct it:

a)
```
K = 10;
while ( K <= 1 );
   --K;
}
```

b)
```
for ( i = 1.0; i != 0.1; i += 0.1 )
   System.out.println( i );
```
c)
```
switch ( m )
{
   case 10:
      System.out.println( "The number is 10" );
   case 20:
      System.out.println( "The number is 20" );
      break;
   default:
      System.out.println( "The number is not 10 or 20" );
      break;
}
```
d) The following code should print the values 1 to 20:
```
m = 1;
while ( m < 20 )
   System.out.println( m++ );
```

Answers to Self-Review Exercises (Sections C.1–C.13)

C.1 a) Pseudocode. b) sequential execution. c) sentinel, signal, flag or dummy. d) strongly typed. e) Repetition statements.

C.2 a) False. It is a single-selection statement because it selects or ignores a *single* action b) False. Java's ternary operator (?:) cannot be used in place of an do…while statement. c) True. d) True. e) False. A nonfatal logic error allows a program to continue executing but causes it to produce incorrect results.

C.3 a) z = x * y--;

b)
```
if ( count == 10 )
   System.out.println( "Count is equal to 10" );
```
c) `y = (x < 20) ? x + 20: x - 20;`

d) `total += x - 5;`

C.4 a)
```
int sum;
int x;
```
b) `x = 1;`

c) `sum = 0;`

d) `sum += x;` or `sum = sum + x;`

e) `System.out.printf( "The sum is: %d\n", sum );`

C.5 `product = 25, x = 6`

C.6 a) Error: The closing right brace of the `while` statement's body is missing.
Correction: Add a closing right brace after the statement `++c;`.

b) Error: The semicolon after `else` results in a logic error. The second output statement will always be executed.
Correction: Remove the semicolon after `else`.

C.7 The value of the variable `z` is never changed in the `while` statement. Therefore, if the loop-continuation condition `( z >= 0 )` is true, an infinite loop is created. To prevent an infinite loop from occurring, `z` must be decremented so that it eventually becomes less than 0.

Answers to Self-Review Exercises (Sections C.14–C.20)

C.8 a) `for`, `while`. b) after. c) `switch`. d) `continue`. e) `&&` (conditional AND). f) `false`.

C.9 a) False. The `default` case is optional. If no default action is needed, then there's no need for a `default` case. b) False. The `break` statement is used to exit the `switch` statement. The `break` statement is not required for the last case in a `switch` statement. c) False. Both of the relational expressions must be true for the entire expression to be true when using the `&&` operator. d) True. e) True.

C.10 a)
```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
   sum += count;
```
b)
```
double result = Math.pow( 2.5, 3 );
```
c)
```
i = 1;

while ( i <= 20 )
{
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( '\t' );

   ++i;
}
```
d)
```
for ( i = 1; i <= 20; ++i )
{
   System.out.print( i );

   if ( i % 5 == 0 )
      System.out.println();
   else
      System.out.print( '\t' );
}
```

C.11
a) Error: The semicolon after the while header causes an infinite loop, and there's a missing left brace.
Correction: Replace the semicolon by a {, or remove both the ; and the }.
b) Error: Using a floating-point number to control a `for` statement may not work, because floating-point numbers are represented only approximately by most computers.
Correction: Use an integer, and perform the proper calculation in order to get the values you desire:
```
for ( i = 1; i != 10; ++i )
   System.out.println( (double) i / 10 );
```
c) Error: The missing code is the `break` statement in the statements for the first `case`.
Correction: Add a `break` statement at the end of the statements for the first `case`. This omission is not necessarily an error if you want the statement of `case 2:` to execute every time the `case 1:` statement executes.
d) Error: An improper relational operator is used in the `while`'s continuation condition.
Correction: Use `<=` rather than `<`, or change `20` to `21`.

Exercises (Sections C.1–C.13)

C.12 Explain what happens when a Java program attempts to divide one integer by another. What happens to the fractional part of the calculation? How can you avoid that outcome?

C.13 Describe the two ways in which control statements can be combined.

C.14 What type of repetition would be appropriate for calculating the sum of the first 100 positive integers? What type would be appropriate for calculating the sum of an arbitrary number of positive integers? Briefly describe how each of these tasks could be performed.

C.15 What is the difference between preincrementing and postincrementing a variable?

C.16 Identify and correct the errors in each of the following pieces of code. [*Note:* There may be more than one error in each piece of code.]

a)
```
if ( age >= 65 );
   System.out.println( "Age is greater than or equal to 65" );
else
   System.out.println( "Age is less than 65 )";
```
b)
```
int x = 1, total;
while ( x <= 10 )
{
   total += x;
   ++x;
}
```
c)
```
while ( x <= 100 )
   total += x;
   ++x;
```
d)
```
while ( y > 0 )
{
   System.out.println( y );
   ++y;
```

For Exercise C.17 and Exercise C.18, perform each of the following steps:

a) Read the problem statement.
b) Write a Java program.
c) Test, debug and execute the Java program.
d) Process three complete sets of data.

C.17 *(Gas Mileage)* Drivers are concerned with the mileage their automobiles get. One driver has kept track of several trips by recording the miles driven and gallons used for each tankful. Develop a Java application that will input the miles driven and gallons used (both as integers) for each trip. The program should calculate and display the miles per gallon obtained for each trip and print the combined miles per gallon obtained for all trips up to this point. All averaging calculations should produce floating-point results. Use class `Scanner` and sentinel-controlled repetition to obtain the data from the user.

C.18 *(Credit Limit Calculator)* Develop a Java application that determines whether any of several department-store customers has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a) account number
b) balance at the beginning of the month
c) total of all items charged by the customer this month
d) total of all credits applied to the customer's account this month
e) allowed credit limit

The program should input all these facts as integers, calculate the new balance (= *beginning balance + charges – credits*), display the new balance and determine whether the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the message `"Credit limit exceeded"`.

C.19 *(Find the Largest Number)* The process of finding the largest value is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program, then a Java application that inputs a series of 10 integers and determines and prints the largest integer. Your program should use at least the following three variables:

a) `counter`: A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).
b) `number`: The integer most recently input by the user.
c) `largest`: The largest number found so far.

C.20 *(Tabular Output)* Write a Java application that uses looping to print the following table of values:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

C.21 *(Multiples of 2 with an Infinite Loop)* Write an application that keeps displaying in the command window the multiples of the integer 2—namely, 2, 4, 8, 16, 32, 64, and so on. Your loop should not terminate (i.e., it should create an infinite loop). What happens when you run this program?

Exercises (Sections C.14–C.20)

C.22 Describe briefly the selection statements available in Java.

C.23 *(Find the Largest Value)* Write an application that finds the largest even number in several integers. Assume that the first value read specifies the number of values to input from the user.

C.24 Assume that `i = 3`, `j = 4`, `k = 3` and `m = 2`. What does each of the following statements print?

a) `System.out.println( i == 1 );`
b) `System.out.println( j == 4 );`
c) `System.out.println( ( i >= 1 ) && ( j == 4 ) );`
d) `System.out.println( ( m != 99 ) & ( k <= m) );`
e) `System.out.println( ( j >= i ) || ( k == m ) );`
f) `System.out.println( ( k + m < j ) | ( 3 - j >= k ) );`
g) `System.out.println( !( k > m ) );`

C.25 *(Calculating the Value of π)* Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the value of π approximated by computing the first 200,000 terms of this series. How many terms do you have to use before you first get a value that begins with 3.14159?

C.26 What does the following program segment do?

```
for ( i = 0; i < 10; ++i )
{
   for ( j = 0; j < 5; ++j )
   {
      for ( k = 0; k < 3; ++k )
         System.out.print(  '#' );

      System.out.println();
   } // end inner for

   System.out.println();
} // end outer for
```

C.27 *("The Twelve Days of Christmas" Song)* Write (as concisely as possible) an application that uses repetition and one or more `switch` statements to print the song "The Twelve Days of Christmas."

D Methods: A Deeper Look

Objectives

In this appendix you'll learn:

- How `static` methods and fields are associated with classes rather than objects.
- How the method call/return mechanism is supported by the method-call stack.
- How packages group related classes.
- To use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.

Outline

D.1 Introduction

In this appendix, we study methods in more depth. You'll see that it's possible to call certain methods, called `static` methods, without the need for an object of the class to exist. You'll learn how to declare a method with more than one parameter. You'll also learn how Java keeps track of which method is currently executing, how local variables of methods are maintained in memory and how a method knows where to return after it completes execution.

We'll take a brief diversion into simulation techniques with random-number generation and develop a version of the casino dice game called craps that uses most of the programming techniques you've used to this point in the book. In addition, you'll learn how to declare values that cannot change (i.e., constants) in your programs.

Many of the classes you'll use or create while developing applications will have more than one method of the same name. This technique, called overloading, is used to implement methods that perform similar tasks for arguments of different types or for different numbers of arguments.

D.2 Program Modules in Java

You write Java programs by combining new methods and classes with predefined ones available in the **Java Application Programming Interface** (also referred to as the **Java API** or **Java class library**) and in various other class libraries. Related classes are typically grouped into *packages* so that they can be imported into programs and reused. You'll learn how to group your own classes into packages in Appendix F. The Java API provides a rich collection of predefined classes that contain methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, database operations, networking operations, file processing, error checking and many other useful tasks.

Software Engineering Observation D.1

Familiarize yourself with the rich collection of classes and methods provided by the Java API (`docs.oracle.com/javase/6/docs/api/`*) and reuse them when possible. This reduces program development time and avoids introducing programming errors.*

Methods (called **functions** or **procedures** in some languages) help you modularize a program by separating its tasks into self-contained units. You've declared methods in every program you've written. The statements in the method bodies are written only once, are hidden from other methods and can be reused from several locations in a program.

One motivation for modularizing a program into methods is the divide-and-conquer approach, which makes program development more manageable by constructing programs from small, simple pieces. Another is **software reusability**—using existing methods as building blocks to create new programs. Often, you can create programs mostly from standardized methods rather than by building customized code. For example, in earlier programs, we did not define how to read data from the keyboard—Java provides these capabilities in the methods of class `Scanner`. A third motivation is to avoid repeating code. Dividing a program into meaningful methods makes the program easier to debug and maintain.

D.3 `static` Methods, `static` Fields and Class `Math`

Although most methods execute in response to method calls on *specific objects*, this is not always the case. Sometimes a method performs a task that does not depend on the contents of any object. Such a method applies to the class in which it's declared as a whole and is known as a `static` method or a **class method**. It's common for classes to contain convenient `static` methods to perform common tasks. For example, recall that we used `static` method `pow` of class `Math` to raise a value to a power in Fig. C.15. To declare a method as `static`, place the keyword `static` before the return type in the method's declaration. For any class imported into your program, you can call the class's `static` methods by specifying the name of the class in which the method is declared, followed by a dot (`.`) and the method name, as in

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

We use various `Math` class methods here to present the concept of `static` methods. Class `Math` provides a collection of methods that enable you to perform common mathematical calculations. For example, you can calculate the square root of `900.0` with the `static` method call

```
Math.sqrt( 900.0 )
```

The preceding expression evaluates to `30.0`. Method `sqrt` takes an argument of type `double` and returns a result of type `double`. To output the value of the preceding method call in the command window, you might write the statement

```
System.out.println( Math.sqrt( 900.0 ) );
```

In this statement, the value that `sqrt` returns becomes the argument to method `println`. There was no need to create a `Math` object before calling method `sqrt`. Also *all* `Math` class methods are `static`—therefore, each is called by preceding its name with the class name `Math` and the dot (`.`) separator.

Software Engineering Observation D.2

Class `Math` is part of the `java.lang` package, which is implicitly imported by the compiler, so it's not necessary to import class `Math` to use its methods.

Method arguments may be constants, variables or expressions. Figure D.1 summarizes several `Math` class methods. In the figure, *x* and *y* are of type `double`.

Method	Description	Example

`abs( x )`	absolute value of *x*	`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0`
		`ceil( -9.8 )` is `-9.0`

`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`

`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828`
		`exp( 2.0 )` is `7.38906`

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

`log( x )`	natural logarithm of *x* (base *e*)	`log( Math.E )` is `1.0`
		`log( Math.E * Math.E )` is `2.0`

`max( x, y )`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is `12.7`
		`max( -2.3, -12.7 )` is `-2.3`

`min( x, y )`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is `2.3`
		`min( -2.3, -12.7 )` is `-12.7`

`pow( x, y )`	*x* raised to the power *y* (i.e., x^y)	`pow( 2.0, 7.0 )` is `128.0`
		`pow( 9.0, 0.5 )` is `3.0`

`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Fig. D.1 | `Math` class methods.

`Math` Class Constants `PI` and `E`

Class `Math` declares two fields that represent commonly used mathematical constants—**`Math.PI`** and **`Math.E`**. `Math.PI` (3.141592653589793) is the ratio of a circle's circumference to its diameter. `Math.E` (2.718281828459045) is the base value for natural logarithms (calculated with `static Math` method `log`). These fields are declared in class `Math` with the modifiers `public`, `final` and `static`. Making them `public` allows you to use these fields in your own classes. Any field declared with keyword **`final`** is *constant*—its value cannot change after the field is initialized. `PI` and `E` are declared `final` because their values never change. Making these fields `static` allows them to be accessed via the class name `Math` and a dot (`.`) separator, just like class `Math`'s methods. Recall from Section B.4 that when each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as an instance variable—each object (instance) of the class has a separate instance of the variable in memory. There are fields for which each object of a class does *not* have a separate instance of the field. That's the case with `static` fields, which are also known as **class variables**. When objects of a class containing `static` fields are created, all the objects of that class share one copy of the class's `static` fields. Together the class variables (i.e., `static` variables) and instance variables represent the fields of a class. You'll learn more about `static` fields in Section F.10.

Why Is Method `main` Declared `static`?

When you execute the Java Virtual Machine (JVM) with the `java` command, the JVM attempts to invoke the `main` method of the class you specify—when no objects of the class have been created. Declaring `main` as `static` allows the JVM to invoke `main` without creating an instance of the class. When you execute your application, you specify its class name as an argument to the command `java`, as in

```
java ClassName argument1 argument2 …
```

The JVM loads the class specified by *ClassName* and uses that class name to invoke method `main`. In the preceding command, *ClassName* is a **command-line argument** to the JVM that tells it which class to execute. Following the *ClassName*, you can also specify a list of `String`s (separated by spaces) as command-line arguments that the JVM will pass to your application. Such arguments might be used to specify options (e.g., a file name) to run the application. As you'll learn in Appendix E, your application can access those command-line arguments and use them to customize the application.

D.4 Declaring Methods with Multiple Parameters

We now consider how to write your own methods with *multiple* parameters. Figure D.2 uses a method called `maximum` to determine and return the largest of three `double` values. In `main`, lines 14–18 prompt the user to enter three `double` values, then read them from the user. Line 21 calls method `maximum` (declared in lines 28–41) to determine the largest of the three values it receives as arguments. When method `maximum` returns the result to line 21, the program assigns `maximum`'s return value to local variable `result`. Then line 24 outputs the maximum value. At the end of this section, we'll discuss the use of operator + in line 24.

```
 1  // Fig. D.2: MaximumFinder.java
 2  // Programmer-declared method maximum with three double parameters.
 3  import java.util.Scanner;
 4
 5  public class MaximumFinder
 6  {
 7     // obtain three floating-point values and locate the maximum value
 8     public static void main( String[] args )
 9     {
10        // create Scanner for input from command window
11        Scanner input = new Scanner( System.in );
12
13        // prompt for and input three floating-point values
14        System.out.print(
15           "Enter three floating-point values separated by spaces: " );
16        double number1 = input.nextDouble(); // read first double
17        double number2 = input.nextDouble(); // read second double
18        double number3 = input.nextDouble(); // read third double
19
20        // determine the maximum value
21        double result = maximum( number1, number2, number3 );
22
```

Fig. D.2 | Programmer-declared method `maximum` with three `double` parameters. (Part 1 of 2.)

```
23          // display maximum value
24          System.out.println( "Maximum is: " + result );
25       } // end main
26
27       // returns the maximum of its three double parameters
28       public static double maximum( double x, double y, double z )
29       {
30          double maximumValue = x; // assume x is the largest to start
31
32          // determine whether y is greater than maximumValue
33          if ( y > maximumValue )
34             maximumValue = y;
35
36          // determine whether z is greater than maximumValue
37          if ( z > maximumValue )
38             maximumValue = z;
39
40          return maximumValue;
41       } // end method maximum
42    } // end class MaximumFinder
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Fig. D.2 | Programmer-declared method `maximum` with three `double` parameters. (Part 2 of 2.)

The `public` *and* `static` *Keywords*

Method `maximum`'s declaration begins with keyword `public` to indicate that the method is "available to the public"—it can be called from methods of other classes. The keyword `static` enables the `main` method (another `static` method) to call `maximum` as shown in line 21 without qualifying the method name with the class name `MaximumFinder`—`static` methods in the same class can call each other directly. Any other class that uses `maximum` must fully qualify the method name with the class name.

Method `maximum`

In `maximum`'s declaration (lines 28–41), line 28 indicates that it returns a `double` value, that the its name is `maximum` and that it requires three `double` parameters (`x`, `y` and `z`) to accomplish its task. Multiple parameters are specified as a comma-separated list. When `maximum` is called (line 21), the parameters `x`, `y` and `z` are initialized with the values of arguments `number1`, `number2` and `number3`, respectively. There must be one argument in the method call for each parameter in the method declaration. Also, each argument must be *consistent* with the type of the corresponding parameter. For example, a `double` parameter can receive values like 7.35, 22 or –0.03456, but not `String`s like `"hello"` nor the `boolean` values `true` or `false`.

To determine the maximum value, we begin with the assumption that parameter `x` contains the largest value, so line 30 declares local variable `maximumValue` and initializes it with the value of parameter `x`. Of course, it's possible that parameter `y` or `z` contains the actual largest value, so we must compare each of these values with `maximumValue`. The `if` statement at lines 33–34 determines whether `y` is greater than `maximumValue`. If so, line 34 assigns `y` to `maximumValue`. The `if` statement at lines 37–38 determines whether `z` is greater than `maximumValue`. If so, line 38 assigns `z` to `maximumValue`. At this point the largest of the three values resides in `maximumValue`, so line 40 returns that value to line 21. When program control returns to the point in the program where `maximum` was called, `maximum`'s parameters `x`, `y` and `z` no longer exist in memory.

Software Engineering Observation D.3

Variables should be declared as fields only if they're required for use in more than one method of the class or if the program should save their values between calls to the class's methods.

Implementing Method `maximum` by Reusing Method `Math.max`

The entire body of our maximum method could also be implemented with two calls to `Math.max`, as follows:

```
      return Math.max( x, Math.max( y, z ) );
```

The first call to `Math.max` specifies arguments `x` and `Math.max( y, z )`. *Before* any method can be called, its arguments must be evaluated to determine their values. If an argument is a method call, the method call must be performed to determine its return value. So, in the preceding statement, `Math.max( y, z )` is evaluated to determine the maximum of `y` and `z`. Then the result is passed as the second argument to the other call to `Math.max`, which returns the larger of its two arguments.

Assembling Strings with String Concatenation

Java allows you to assemble `String` objects into larger strings by using operators + or +=. This is known as **string concatenation**. When both operands of operator + are `String` objects, operator + creates a new `String` object in which the characters of the right operand are placed at the end of those in the left operand—e.g., the expression `"hello " + "there"` creates the `String "hello there"`.

In line 24 of Fig. D.2, the expression `"Maximum is: " + result` uses operator + with operands of types `String` and `double`. *Every primitive value and object in Java has a `String` representation.* When one of the + operator's operands is a `String`, the other is converted to a `String`, then the two are *concatenated*. In line 24, the `double` value is converted to its `String` representation and placed at the end of the `String "Maximum is: "`. If there are any *trailing zeros* in a `double` value, these will be *discarded* when the number is converted to a `String`—for example 9.3500 would be represented as 9.35.

Primitive values used in `String` concatenation are converted to `String`s. A `boolean` concatenated with a `String` is converted to the `String "true"` or `"false"`. All objects have a `toString` method that returns a `String` representation of the object. When an object is concatenated with a `String`, the object's `toString` method is implicitly called to obtain the `String` representation of the object. `ToString` can be called explicitly.

Common Programming Error D.1
It's a syntax error to break a `String` literal across lines. If necessary, you can split a `String` into several smaller `String`s and use concatenation to form the desired `String`.

Common Programming Error D.2
Confusing the + operator used for string concatenation with the + operator used for addition can lead to strange results. Java evaluates the operands of an operator from left to right. For example, if integer variable `y` has the value `5`, the expression `"y + 2 = " + y + 2` results in the string `"y + 2 = 52"`, not `"y + 2 = 7"`, because first the value of `y` (5) is concatenated to the string `"y + 2 = "`, then the value `2` is concatenated to the new larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the desired result `"y + 2 = 7"`.

D.5 Notes on Declaring and Using Methods

There are three ways to call a method:

1. Using a method name by itself to call another method of the *same* class—such as `maximum(number1, number2, number3)` in line 21 of Fig. D.2.
2. Using a variable that contains a reference to an object, followed by a dot (`.`) and the method name to call a non-`static` method of the referenced object—such as the method call in line 13 of Fig. C.3, `myGradeBook.displayMessage()`, which calls a method of class `GradeBook` from the `main` method of `GradeBookTest`.
3. Using the class name and a dot (`.`) to call a `static` method of a class—such as `Math.sqrt(900.0)` in Section D.3.

A `static` method can call *only* other `static` methods of the same class directly (i.e., using the method name by itself) and can manipulate *only* `static` variables in the same class directly. To access the class's non-`static` members, a `static` method must use a reference to an object of the class. Many objects of a class, each with its own copies of the instance variables, may exist at the same time. Suppose a `static` method were to invoke a non-`static` method directly. How would the method know which object's instance variables to manipulate? What would happen if no objects of the class existed at the time the non-`static` method was invoked? Thus, Java does not allow a `static` method to access non-`static` members of the same class directly.

There are three ways to return control to the statement that calls a method. If the method does not return a result, control returns when the program flow reaches the method-ending right brace or when the statement

```
return;
```

is executed. If the method returns a result, the statement

```
return expression;
```

evaluates the *expression*, then returns the result to the caller.

Common Programming Error D.3
Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.

Common Programming Error D.4

Redeclaring a parameter as a local variable in the method's body is a compilation error.

D.6 Method-Call Stack and Activation Records

To understand how Java performs method calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**. You can think of a stack as analogous to a pile of dishes. When a dish is placed on the pile, it's normally placed at the top (referred to as **pushing** the dish onto the stack). Similarly, when a dish is removed from the pile, it's always removed from the top (referred to as **popping** the dish off the stack). Stacks are known as **last-in, first-out (LIFO) data structures**—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.

When a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the **program-execution stack** (sometimes referred to as the **method-call stack**). If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that each method can return to its caller.

The program-execution stack also contains the memory for the local variables used in each invocation of a method during a program's execution. This data, stored as a portion of the program-execution stack, is known as the **activation record** or **stack frame** of the method call. When a method call is made, the activation record for that method call is pushed onto the program-execution stack. When the method returns to its caller, the activation record for this method call is popped off the stack and those local variables are no longer known to the program. If a local variable holding a reference to an object is the only variable in the program with a reference to that object, then, when the activation record containing that local variable is popped off the stack, the object can no longer be accessed by the program and will eventually be deleted from memory by the JVM during "garbage collection." We discuss garbage collection in Section F.9.

Of course, a computer's memory is finite, so only a certain amount can be used to store activation records on the program-execution stack. If more method calls occur than can have their activation records stored, an error known as a **stack overflow** occurs.

D.7 Argument Promotion and Casting

Another important feature of method calls is **argument promotion**—converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter. For example, a program can call `Math` method `sqrt` with an `int` argument even though a `double` argument is expected. The statement

```
    System.out.println( Math.sqrt( 4 ) );
```

correctly evaluates `Math.sqrt(4)` and prints the value `2.0`. The method declaration's parameter list causes Java to convert the `int` value `4` to the `double` value `4.0` before passing the value to method `sqrt`. Such conversions may lead to compilation errors if Java's **promotion rules** are not satisfied. These rules specify which conversions are allowed—that is, which ones can be performed without losing data. In the `sqrt` example above, an `int` is converted to a `double` without changing its value. However, converting a `double` to an `int` truncates the fractional part of the `double` value—thus, part of the value is lost. Converting large integer types to small integer types (e.g., `long` to `int`, or `int` to `short`) may also result in changed values.

The promotion rules apply to expressions containing values of two or more primitive types and to primitive-type values passed as arguments to methods. Each value is promoted to the "highest" type in the expression. Actually, the expression uses a temporary copy of each value—the types of the original values remain unchanged. Figure D.3 lists the primitive types and the types to which each can be promoted. The valid promotions for a given type are always to a type higher in the table. For example, an `int` can be promoted to the higher types `long`, `float` and `double`.

Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Fig. D.3 | Promotions allowed for primitive types.

Converting values to types lower in the table of Fig. D.3 will result in different values if the lower type cannot represent the value of the higher type (e.g., the `int` value 2000000 cannot be represented as a `short`, and any floating-point number with digits after its decimal point cannot be represented in an integer type such as `long`, `int` or `short`). Therefore, in cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator (introduced in Section C.9) to explicitly force the conversion to occur—otherwise a compilation error occurs. This enables you to "take control" from the compiler. You essentially say, "I know this conversion might cause loss of information, but for my purposes here, that's fine." Suppose method `square` calculates the square of an integer and thus requires an `int` argument. To call `square` with a `double` argument named `doubleValue`, we would be required to write the method call as

```
    square( (int) doubleValue )
```

This method call explicitly casts (converts) a *copy* of variable `doubleValue`'s value to an integer for use in method `square`. Thus, if `doubleValue`'s value is `4.5`, the method receives the value `4` and returns `16`, not `20.25`.

D.8 Java API Packages

As you've seen, Java contains many predefined classes that are grouped into categories of related classes called packages. Together, these are known as the Java Application Programming Interface (Java API), or the Java class library. A great strength of Java is the Java

API's thousands of classes. Some key Java API packages used in this book's appendices are described in Fig. D.4, which represents only a small portion of the reusable components in the Java API.

Package	Description
`java.awt.event`	The **Java Abstract Window Toolkit Event Package** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages.
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data.
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed bookwide) that are required by many Java programs. This package is imported by the compiler into all programs.
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`) and the storing and processing of large amounts of data.
`java.util.concurrent`	The **Java Concurrency Package** contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel.
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs.

Fig. D.4 | Java API packages (a subset).

The set of packages available in Java is quite large. In addition to those summarized in Fig. D.4, Java includes packages for complex graphics, advanced graphical user interfaces, printing, advanced networking, security, database processing, multimedia, accessibility (for people with disabilities), concurrent programming, cryptography, XML processing and many other capabilities. Many other packages are also available for download at `java.sun.com`.

You can locate additional information about a predefined Java class's methods in the Java API documentation at `docs.oracle.com/javase/6/docs/api/`. When you visit this site, click the **Index** link to see an alphabetical listing of all the classes and methods in the Java API. Locate the class name and click its link to see the online description of the class. Click the **METHOD** link to see a table of the class's methods. Each `static` method will be listed with the word "`static`" preceding its return type.

D.9 Introduction to Random-Number Generation

We now take a brief diversion into a popular type of programming application—simulation and game playing. In this and the next section, we develop a nicely structured game-playing program with multiple methods. The program uses most of the control statements presented thus far in the appendices and introduces several new programming concepts.

Random numbers can be introduced in a program via an object of class **`Random`** (package `java.util`) or via the `static` method `random` of class `Math`. A `Random` object can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values, whereas `Math` method `random` can produce only `double` values in the range $0.0 \le x < 1.0$, where x is the value returned by method `random`. In the next several examples, we use objects of class `Random` to produce random values. We discuss only random `int` values here. For more information on the `Random` class, see `docs.oracle.com/javase/6/docs/api/java/util/Random.html`.

A new random-number generator object can be created as follows:

```
      Random randomNumbers = new Random();
```

Consider the following statement:

```
      int randomValue = randomNumbers.nextInt();
```

`Random` method **`nextInt`** generates a random `int` value in the range –2,147,483,648 to +2,147,483,647, inclusive. If it truly produces values at random, then every value in the range should have an equal chance (or probability) of being chosen each time `nextInt` is called. The numbers are actually **pseudorandom numbers**—a sequence of values produced by a complex mathematical calculation. The calculation uses the current time of day (which, of course, changes constantly) to **seed** the random-number generator such that each execution of a program yields a different sequence of random values.

The range of values produced directly by method `nextInt` generally differs from the range of values required in a particular Java application. For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A program that simulates the rolling of a six-sided die might require random integers in the range 1–6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1–4. For cases like these, class `Random` provides another version of method `nextInt` that receives an `int` argument and returns a value from 0 up to, but not including, the argument's value. For example, for coin tossing, the following statement returns 0 or 1.

```
      int randomValue = randomNumbers.nextInt( 2 );
```

D.9.1 Scaling and Shifting of Random Numbers

To demonstrate random numbers, let's show to simulate rolling a six-sided die. We begin by using `nextInt` to produce random values in the range 0–5, as follows:

```
      face = randomNumbers.nextInt( 6 );
```

The argument `6`—called the **scaling factor**—represents the number of unique values that `nextInt` should produce (in this case six—0, 1, 2, 3, 4 and 5). This manipulation is called **scaling** the range of values produced by `Random` method `nextInt`.

A six-sided die has the numbers 1–6 on its faces, not 0–5. So we **shift** the range of numbers produced by adding a **shifting value**—in this case 1—to our previous result, as in

```
      face = 1 + randomNumbers.nextInt( 6 );
```

The shifting value (`1`) specifies the *first* value in the desired range of random integers. The preceding statement assigns `face` a random integer in the range 1–6. The numbers produced by `nextInt` occur with approximately equal likelihood.

Generalizing the Random Number Calculations

The preceding statement always assigns to variable `face` an integer in the range `1 ≤ face ≤ 6`. The width of this range (i.e., the number of consecutive integers in the range) is `6`, and the starting number in the range is `1`. The width of the range is determined by the number `6` that's passed as an argument to `Random` method `nextInt`, and the starting number of the range is the number `1` that's added to the result of calling `nextInt`. We can generalize this result as

```
      number = shiftingValue + randomNumbers.nextInt( scalingFactor );
```

where *shiftingValue* specifies the first number in the desired range of consecutive integers and *scalingFactor* specifies how many numbers are in the range.

It's also possible to choose integers at random from sets of values other than ranges of consecutive integers. For example, to obtain a random value from the sequence 2, 5, 8, 11 and 14, you could use the statement

```
      number = 2 + 3 * randomNumbers.nextInt( 5 );
```

In this case, `randomNumbers.nextInt(5)` produces values in the range 0–4. Each value produced is multiplied by 3 to produce a number in the sequence 0, 3, 6, 9 and 12. We add 2 to that value to shift the range of values and obtain a value from the sequence 2, 5, 8, 11 and 14. We can generalize this result as

```
      number = shiftingValue +
         differenceBetweenValues * randomNumbers.nextInt( scalingFactor );
```

where *shiftingValue* specifies the first number in the desired range of values, *difference-BetweenValues* represents the constant difference between consecutive numbers in the sequence and *scalingFactor* specifies how many numbers are in the range.

D.9.2 Random-Number Repeatability for Testing and Debugging

Class `Random`'s methods actually generate pseudorandom numbers based on complex mathematical calculations—the sequence of numbers appears to be random. The calculation that produces the numbers uses the time of day as a **seed value** to change the sequence's starting point. Each new `Random` object seeds itself with a value based on the computer system's clock at the time the object is created, enabling each execution of a program to produce a different sequence of random numbers.

When debugging an application, it's often useful to repeat the exact same sequence of pseudorandom numbers during each execution of the program. This repeatability enables you to prove that your application is working for a specific sequence of random numbers before you test it with different sequences of random numbers. When repeatability is important, you can create a `Random` object as follows:

```
      Random randomNumbers = new Random( seedValue );
```

The `seedValue` argument (of type `long`) seeds the random-number calculation. If the same `seedValue` is used every time, the `Random` object produces the same sequence of numbers. You can set a `Random` object's seed at any time during program execution by calling the object's `set` method, as in

```
      randomNumbers.set( seedValue );
```

Error-Prevention Tip D.1

While developing a program, create the `Random` object with a specific seed value to produce a repeatable sequence of numbers each time the program executes. If a logic error occurs, fix the error and test the program again with the same seed value—this allows you to reconstruct the same sequence of numbers that caused the error. Once the logic errors have been removed, create the `Random` object without using a seed value, causing the `Random` object to generate a new sequence of random numbers each time the program executes.

D.10 Case Study: A Game of Chance; Introducing Enumerations

A popular game of chance is a dice game known as craps, which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

> *You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called "craps"), you lose (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.*

Figure D.5 simulates the game of craps, using methods to implement the game's logic. The `main` method (lines 21–65) calls the `rollDice` method (lines 68–81) as necessary to roll the dice and compute their sum. The sample outputs show winning and losing on the first roll, and winning and losing on a subsequent roll.

```
 1  // Fig. D.5: Craps.java
 2  // Craps class simulates the dice game craps.
 3  import java.util.Random;
 4
 5  public class Craps
 6  {
 7     // create random number generator for use in method rollDice
 8     private static final Random randomNumbers = new Random();
 9
10     // enumeration with constants that represent the game status
11     private enum Status { CONTINUE, WON, LOST };
12
13     // constants that represent common rolls of the dice
14     private static final int SNAKE_EYES = 2;
15     private static final int TREY = 3;
16     private static final int SEVEN = 7;
17     private static final int YO_LEVEN = 11;
18     private static final int BOX_CARS = 12;
19
20     // plays one game of craps
21     public static void main( String[] args )
22     {
23        int myPoint = 0; // point if no win or loss on first roll
```

Fig. D.5 | `Craps` class simulates the dice game craps. (Part 1 of 3.)

```
      Status gameStatus; // can contain CONTINUE, WON or LOST

      int sumOfDice = rollDice(); // first roll of the dice

      // determine game status and point based on first roll
      switch ( sumOfDice )
      {
         case SEVEN: // win with 7 on first roll
         case YO_LEVEN: // win with 11 on first roll
            gameStatus = Status.WON;
            break;
         case SNAKE_EYES: // lose with 2 on first roll
         case TREY: // lose with 3 on first roll
         case BOX_CARS: // lose with 12 on first roll
            gameStatus = Status.LOST;
            break;
         default: // did not win or lose, so remember point
            gameStatus = Status.CONTINUE; // game is not over
            myPoint = sumOfDice; // remember the point
            System.out.printf( "Point is %d\n", myPoint );
            break; // optional at end of switch
      } // end switch

      // while game is not complete
      while ( gameStatus == Status.CONTINUE ) // not WON or LOST
      {
         sumOfDice = rollDice(); // roll dice again

         // determine game status
         if ( sumOfDice == myPoint ) // win by making point
            gameStatus = Status.WON;
         else
            if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
               gameStatus = Status.LOST;
      } // end while

      // display won or lost message
      if ( gameStatus == Status.WON )
         System.out.println( "Player wins" );
      else
         System.out.println( "Player loses" );
   } // end main

   // roll dice, calculate sum and display results
   public static int rollDice()
   {
      // pick random die values
      int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
      int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll

      int sum = die1 + die2; // sum of die values

```

Fig. D.5 | Craps class simulates the dice game craps. (Part 2 of 3.)

```
76         // display results of this roll
77         System.out.printf( "Player rolled %d + %d = %d\n",
78            die1, die2, sum );
79
80         return sum; // return sum of dice
81      } // end method rollDice
82   } // end class Craps
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

Fig. D.5 | `Craps` class simulates the dice game craps. (Part 3 of 3.)

Method `rollDice`

In the rules of the game, the player must roll two dice on the first roll and must do the same on all subsequent rolls. We declare method `rollDice` (Fig. D.5, lines 68–81) to roll the dice and compute and print their sum. Method `rollDice` is declared once, but it's called from two places (lines 26 and 50) in `main`, which contains the logic for one complete game of craps. Method `rollDice` takes no arguments, so it has an empty parameter list. Each time it's called, `rollDice` returns the sum of the dice, so the return type `int` is indicated in the method header (line 68). Although lines 71 and 72 look the same (except for the die names), they do not necessarily produce the same result. Each of these statements produces a random value in the range 1–6. Variable `randomNumbers` (used in lines 71–72) is *not* declared in the method. Instead it's declared as a `private static final` variable of the class and initialized in line 8. This enables us to create one `Random` object that's reused in each call to `rollDice`. If there were a program that contained multiple instances of class `Craps`, they'd all share this one `Random` object.

Method `main`*'s Local Variables*

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Method `main` (lines 21–65) uses local variable `myPoint` (line 23) to store the "point" if the player does not win or lose on the first roll, local variable gameStatus (line 24) to keep track of the overall game status and local variable sumOfDice (line 26) to hold the sum of the dice for the most recent roll. Variable myPoint is initialized to 0 to ensure that the application will compile. If you do not initialize myPoint, the compiler issues an error, because myPoint is not assigned a value in *every* case of the switch statement, and thus the program could try to use myPoint before it's assigned a value. By contrast, gameStatus *is* assigned a value in *every* case of the switch statement—thus, it's guaranteed to be initialized before it's used and does not need to be initialized.

enum Type Status

Local variable gameStatus (line 24) is declared to be of a new type called Status (declared at line 11). Type Status is a private member of class Craps, because Status will be used only in that class. Status is a type called an **enumeration**, which, in its simplest form, declares a set of constants represented by identifiers. An enumeration is a special kind of class that's introduced by the keyword **enum** and a type name (in this case, Status). As with classes, braces delimit an enum declaration's body. Inside the braces is a comma-separated list of **enumeration constants**, each representing a unique value. The identifiers in an enum must be unique. You'll learn more about enumerations in Appendix F.

Good Programming Practice D.1

It's a convention to use only uppercase letters in the names of enumeration constants. This makes them stand out and reminds you that they are not variables.

Variables of type Status can be assigned only the three constants declared in the enumeration (line 11) or a compilation error will occur. When the game is won, the program sets local variable gameStatus to Status.WON (lines 33 and 54). When the game is lost, the program sets local variable gameStatus to Status.LOST (lines 38 and 57). Otherwise, the program sets local variable gameStatus to Status.CONTINUE (line 41) to indicate that the game is not over and the dice must be rolled again.

Good Programming Practice D.2

Using enumeration constants (like Status.WON, Status.LOST and Status.CONTINUE) rather than literal values (such as 0, 1 and 2) makes programs easier to read and maintain.

Logic of the main Method

Line 26 in main calls rollDice, which picks two random values from 1 to 6, displays the values of the first die, the second die and their sum, and returns the sum. Method main next enters the switch statement (lines 29–45), which uses the sumOfDice value from line 26 to determine whether the game has been won or lost, or should continue with another roll. The values that result in a win or loss on the first roll are declared as public static final int constants in lines 14–18. The identifier names use casino parlance for these sums. These constants, like enum constants, are declared by convention with all capital letters, to make them stand out in the program. Lines 31–34 determine whether the player won on the first roll with SEVEN (7) or YO_LEVEN (11). Lines 35–39 determine whether the player lost on the first roll with SNAKE_EYES (2), TREY (3), or BOX_CARS (12). After the first roll, if the game is not over, the default case (lines 40–44) sets gameStatus to Status.CONTINUE, saves sumOfDice in myPoint and displays the point.

If we're still trying to "make our point" (i.e., the game is continuing from a prior roll), lines 48–58 execute. Line 50 rolls the dice again. If sumOfDice matches myPoint (line 53), line 54 sets gameStatus to Status.WON, then the loop terminates because the game is complete. If sumOfDice is SEVEN (line 56), line 57 sets gameStatus to Status.LOST, and the loop terminates because the game is complete. When the game completes, lines 61–64 display a message indicating whether the player won or lost, and the program terminates.

The program uses the various program-control mechanisms we've discussed. The Craps class uses two methods—main and rollDice (called twice from main)—and the switch, while, if...else and nested if control statements. Note also the use of multiple case labels in the switch statement to execute the same statements for sums of SEVEN and YO_LEVEN (lines 31–32) and for sums of SNAKE_EYES, TREY and BOX_CARS (lines 35–37).

Why Some Constants Are Not Defined as **enum** *Constants*

You might be wondering why we declared the sums of the dice as public final static int constants rather than as enum constants. The reason is that the program must compare the int variable sumOfDice (line 26) to these constants to determine the outcome of each roll. Suppose we declared enum Sum containing constants (e.g., Sum.SNAKE_EYES) representing the five sums used in the game, then used these constants in the switch statement (lines 29–45). Doing so would prevent us from using sumOfDice as the switch statement's controlling expression, because Java does *not* allow an int to be compared to an enumeration constant. To achieve the same functionality as the current program, we would have to use a variable currentSum of type Sum as the switch's controlling expression. Unfortunately, Java does not provide an easy way to convert an int value to a particular enum constant. This could be done with a separate switch statement. Clearly this would be cumbersome and not improve the program's readability (thus defeating the purpose of using an enum).

D.11 Scope of Declarations

You've seen declarations of various Java entities, such as classes, methods, variables and parameters. Declarations introduce names that can be used to refer to such Java entities. The **scope** of a declaration is the portion of the program that can refer to the declared entity by its name. Such an entity is said to be "in scope" for that portion of the program. This section introduces several important scope issues.

The basic scope rules are as follows:

1. The scope of a parameter declaration is the body of the method in which the declaration appears.
2. The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.
3. The scope of a local-variable declaration that appears in the initialization section of a for statement's header is the body of the for statement and the other expressions in the header.
4. A method or field's scope is the entire body of the class. This enables non-static methods of a class to use the fields and other methods of the class.

Any block may contain variable declarations. If a local variable or parameter in a method has the same name as a field of the class, the field is "hidden" until the block terminates execution—this is called **shadowing**. In Appendix F, we discuss how to access shadowed fields.

Error-Prevention Tip D.2

Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method shadows a field in the class.

Figure D.6 demonstrates scoping issues with fields and local variables. Line 7 declares and initializes the field `x` to `1`. This field is shadowed (hidden) in any block (or method) that declares a local variable named `x`. Method `main` (lines 11–23) declares a local variable `x` (line 13) and initializes it to `5`. This local variable's value is output to show that the field `x` (whose value is `1`) is shadowed in `main`. The program declares two other methods—`useLocalVariable` (lines 26–35) and `useField` (lines 38–45)—that each take no arguments and return no results. Method `main` calls each method twice (lines 17–20). Method `useLocalVariable` declares local variable `x` (line 28). When `useLocalVariable` is first called (line 17), it creates local variable `x` and initializes it to `25` (line 28), outputs the value of `x` (lines 30–31), increments `x` (line 32) and outputs the value of `x` again (lines 33–34). When `uselLocalVariable` is called a second time (line 19), it recreates local variable `x` and reinitializes it to `25`, so the output of each `useLocalVariable` call is identical.

```
 1  // Fig. D.6: Scope.java
 2  // Scope class demonstrates field and local variable scopes.
 3
 4  public class Scope
 5  {
 6     // field that is accessible to all methods of this class
 7     private static int x = 1;
 8
 9     // method main creates and initializes local variable x
10     // and calls methods useLocalVariable and useField
11     public static void main( String[] args )
12     {
13        int x = 5; // method's local variable x shadows field x
14
15        System.out.printf( "local x in main is %d\n", x );
16
17        useLocalVariable(); // useLocalVariable has local x
18        useField(); // useField uses class Scope's field x
19        useLocalVariable(); // useLocalVariable reinitializes local x
20        useField(); // class Scope's field x retains its value
21
22        System.out.printf( "\nlocal x in main is %d\n", x );
23     } // end main
24
25     // create and initialize local variable x during each call
26     public static void useLocalVariable()
27     {
28        int x = 25; // initialized each time useLocalVariable is called
29
30        System.out.printf(
31           "\nlocal x on entering method useLocalVariable is %d\n", x );
32        ++x; // modifies this method's local variable x
```

Fig. D.6 | `Scope` class demonstrates field and local variable scopes. (Part 1 of 2.)

```
33         System.out.printf(
34            "local x before exiting method useLocalVariable is %d\n", x );
35      } // end method useLocalVariable
36
37      // modify class Scope's field x during each call
38      public static void useField()
39      {
40         System.out.printf(
41            "\nfield x on entering method useField is %d\n", x );
42         x *= 10; // modifies class Scope's field x
43         System.out.printf(
44            "field x before exiting method useField is %d\n", x );
45      } // end method useField
46   } // end class Scope
```

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

Fig. D.6 | `Scope` class demonstrates field and local variable scopes. (Part 2 of 2.)

Method `useField` does not declare any local variables. Therefore, when it refers to `x`, field `x` (line 7) of the class is used. When method `useField` is first called (line 18), it outputs the value (`1`) of field `x` (lines 40–41), multiplies the field `x` by `10` (line 42) and outputs the value (`10`) of field `x` again (lines 43–44) before returning. The next time method `useField` is called (line 20), the field has its modified value (`10`), so the method outputs `10`, then `100`. Finally, in method `main`, the program outputs the value of local variable `x` again (line 22) to show that none of the method calls modified `main`'s local variable `x`, because the methods all referred to variables named `x` in other scopes.

D.12 Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters)—this is called **method overloading**. When an overloaded method is called, the compiler selects the appropriate method by examining the number, types and order of the arguments in the call. Method overloading is commonly used to create several methods with the *same* name that perform the *same* or *similar* tasks, but on different types or different numbers of arguments. For example, `Math` methods `abs`, `min` and `max` (summarized in Section D.3) are overloaded with four versions each:

1. One with two `double` parameters.
2. One with two `float` parameters.
3. One with two `int` parameters.
4. One with two `long` parameters.

Our next example demonstrates declaring and invoking overloaded methods. We demonstrate overloaded constructors in Appendix F.

Declaring Overloaded Methods

Class `MethodOverload` (Fig. D.7) includes two overloaded versions of method `square`—one that calculates the square of an `int` (and returns an `int`) and one that calculates the square of a `double` (and returns a `double`). Although these methods have the same name and similar parameter lists and bodies, think of them simply as *different* methods. It may help to think of the method names as "`square` of `int`" and "`square` of `double`," respectively.

```java
// Fig. D.7: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public static void main( String[] args )
   {
      System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
      System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
   } // end main

   // square method with int argument
   public static int square( int intValue )
   {
      System.out.printf( "\nCalled square with int argument: %d\n",
         intValue );
      return intValue * intValue;
   } // end method square with int argument

   // square method with double argument
   public static double square( double doubleValue )
   {
      System.out.printf( "\nCalled square with double argument: %f\n",
         doubleValue );
      return doubleValue * doubleValue;
   } // end method square with double argument
} // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. D.7 | Overloaded method declarations.

Line 9 invokes method `square` with the argument `7`. Literal integer values are treated as type `int`, so the method call in line 9 invokes the version of `square` at lines 14–19 that specifies an `int` parameter. Similarly, line 10 invokes method `square` with the argument `7.5`. Literal floating-point values are treated as type `double`, so the method call in line 10 invokes the version of `square` at lines 22–27 that specifies a `double` parameter. Each method first outputs a line of text to prove that the proper method was called in each case. The values in lines 10 and 24 are displayed with the format specifier `%f`. We did not specify a precision in either case. By default, floating-point values are displayed with six digits of precision if the precision is not specified in the format specifier.

Distinguishing Between Overloaded Methods

The compiler distinguishes overloaded methods by their **signature**—a combination of the method's name and the number, types and order of its parameters. If the compiler looked only at method names during compilation, the code in Fig. D.7 would be ambiguous—the compiler would not know how to distinguish between the two `square` methods (lines 14–19 and 22–27). Internally, the compiler uses longer method names that include the original method name, the types of each parameter and the exact order of the parameters to determine whether the methods in a class are unique in that class.

For example, in Fig. D.7, the compiler might use the logical name "`square` of `int`" for the `square` method that specifies an `int` parameter and "`square` of `double`" for the `square` method that specifies a `double` parameter (the actual names the compiler uses are messier). If `method1`'s declaration begins as

```
void method1( int a, float b )
```

then the compiler might use the logical name "`method1` of `int` and `float`." If the parameters are specified as

```
void method1( float a, int b )
```

then the compiler might use the logical name "`method1` of `float` and `int`." The *order* of the parameter types is important—the compiler considers the preceding two `method1` headers to be distinct.

Return Types of Overloaded Methods

In discussing the logical names of methods used by the compiler, we did not mention the return types of the methods. *Method calls cannot be distinguished by return type*. If you had overloaded methods that differed only by their return types and you called one of the methods in a standalone statement as in:

```
square( 2 );
```

the compiler would *not* be able to determine the version of the method to call, because the return value is ignored. When two methods have the same signature and different return types, the compiler issues an error message indicating that the method is already defined in the class. Overloaded methods *can* have different return types if the methods have different parameter lists. Also, overloaded methods need *not* have the same number of parameters.

Common Programming Error D.5

Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.

D.13 Wrap-Up

In this appendix, you learned more about method declarations. You also learned the difference between non-`static` and `static` methods and how to call `static` methods by preceding the method name with the name of the class in which it appears and the dot (`.`) separator. You learned how to use operators `+` and `+=` to perform string concatenations. We discussed how the method-call stack and activation records keep track of the methods that have been called and where each method must return to when it completes its task. We also discussed Java's promotion rules for converting implicitly between primitive types and how to perform explicit conversions with cast operators. Next, you learned about some of the commonly used packages in the Java API.

You saw how to declare named constants using both `enum` types and `public static final` variables. You used class `Random` to generate random numbers for simulations. You also learned about the scope of fields and local variables in a class. Finally, you learned that multiple methods in one class can be overloaded by providing methods with the same name and different signatures. Such methods can be used to perform the same or similar tasks using different types or different numbers of parameters.

In Appendix E, you'll learn how to maintain lists and tables of data in arrays. You'll see a more elegant implementation of the application that rolls a die 6,000,000 times and two enhanced versions of our `GradeBook` case study that you studied in Appendices B–C. You'll also learn how to access an application's command-line arguments that are passed to method `main` when an application begins execution.

Self-Review Exercises

D.1 Fill in the blanks in each of the following statements:

a) A method is invoked with a(n) ________.
b) A variable known only within the method in which it's declared is called a(n) ________.
c) ________ class provides a collection of methods that enable you to perform common mathematical calculations.
d) ________ variables do not have separate instances for each object of a class.
e) Data can be added or removed only from the ________ of a stack.
f) Stacks are known as ________ data structures; the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.
g) The three ways to return control from a called method to a caller are ________, ________ and ________.
h) An object of class ________ produces random numbers.
i) The program-execution stack contains the memory for local variables on each invocation of a method during a program's execution. This data, stored as a portion of the program-execution stack, is known as the ________ or ________ of the method call.
j) If there are more method calls than can be stored on the program-execution stack, an error known as a(n) ________ occurs.
k) The ________ of a declaration is the portion of a program that can refer to the entity in the declaration by name.
l) It's possible to have several methods with the same name that each operate on different types or numbers of arguments. This feature is called method ________.
m) The program-execution stack is also referred to as the ________ stack.

D.2 For the class `Craps` in Fig. D.5, state the scope of each of the following entities:

a) the variable `randomNumbers`.
b) the variable `die1`.
c) the method `rollDice`.
d) the method `main`.
e) the variable `sumOfDice`.

D.3 Write an application that tests whether the examples of the `Math` class method calls shown in Fig. D.1 actually produce the indicated results.

D.4 Give the method header for each of the following methods:

a) Method `hypotenuse`, which takes two double-precision, floating-point arguments `side1` and `side2` and returns a double-precision, floating-point result.
b) Method `smallest`, which takes three integers `x`, `y` and `z` and returns an integer.
c) Method `instructions`, which does not take any arguments and does not return a value. [*Note:* Such methods are commonly used to display instructions to a user.]
d) Method `intToFloat`, which takes integer argument `number` and returns a `float`.

D.5 Find the error in each of the following program segments. Explain how to correct the error.

a)
```
void g()
{
   System.out.println( "Inside method g" );

   void h()
   {
      System.out.println( "Inside method h" );
   }
}
```
b)
```
int sum( int x, int y )
{
   int result;
   result = x + y;
}
```
c)
```
void f( float a );
{
   float a;
   System.out.println( a );
}
```

D.6 Write a complete Java application to prompt the user for the double radius of a circle, and call method `circleArea` to calculate and display the area of the circle. Use the following statement to calculate the area:

```
double area =  Math.PI * Math.pow( radius, 2 )
```

Answers to Self-Review Exercises

D.1 a) method call. b) local variable. c) `Math`. d) Class. e) top. f) last-in, first-out (LIFO). g) `return;` or `return` *expression*; or encountering the closing right brace of a method. h) `Random`. i) activation record, stack frame. j) stack overflow. k) scope. l) method overloading. m) method call.

D.2 a) class body. b) block that defines method `rollDice`'s body. c) class body. d) class body. e) block that defines method `main`'s body.

D.3 The following solution demonstrates the `Math` class methods in Fig. D.1:

```java
// Exercise D.3: MathTest.java
// Testing the Math class methods.

public class MathTest
{
   public static void main( String[] args )
   {
      System.out.printf( "Math.abs( 23.7 ) = %f\n", Math.abs( 23.7 ) );
      System.out.printf( "Math.abs( 0.0 ) = %f\n", Math.abs( 0.0 ) );
      System.out.printf( "Math.abs( -23.7 ) = %f\n", Math.abs( -23.7 ) );
      System.out.printf( "Math.ceil( 9.2 ) = %f\n", Math.ceil( 9.2 ) );
      System.out.printf( "Math.ceil( -9.8 ) = %f\n", Math.ceil( -9.8 ) );
      System.out.printf( "Math.cos( 0.0 ) = %f\n", Math.cos( 0.0 ) );
      System.out.printf( "Math.exp( 1.0 ) = %f\n", Math.exp( 1.0 ) );
      System.out.printf( "Math.exp( 2.0 ) = %f\n", Math.exp( 2.0 ) );
      System.out.printf( "Math.floor( 9.2 ) = %f\n", Math.floor( 9.2 ) );
      System.out.printf( "Math.floor( -9.8 ) = %f\n",
         Math.floor( -9.8 ) );
      System.out.printf( "Math.log( Math.E ) = %f\n",
         Math.log( Math.E ) );
      System.out.printf( "Math.log( Math.E * Math.E ) = %f\n",
         Math.log( Math.E * Math.E ) );
      System.out.printf( "Math.max( 2.3, 12.7 ) = %f\n",
         Math.max( 2.3, 12.7 ) );
      System.out.printf( "Math.max( -2.3, -12.7 ) = %f\n",
         Math.max( -2.3, -12.7 ) );
      System.out.printf( "Math.min( 2.3, 12.7 ) = %f\n",
         Math.min( 2.3, 12.7 ) );
      System.out.printf( "Math.min( -2.3, -12.7 ) = %f\n",
         Math.min( -2.3, -12.7 ) );
      System.out.printf( "Math.pow( 2.0, 7.0 ) = %f\n",
         Math.pow( 2.0, 7.0 ) );
      System.out.printf( "Math.pow( 9.0, 0.5 ) = %f\n",
         Math.pow( 9.0, 0.5 ) );
      System.out.printf( "Math.sin( 0.0 ) = %f\n", Math.sin( 0.0 ) );
      System.out.printf( "Math.sqrt( 900.0 ) = %f\n",
         Math.sqrt( 900.0 ) );
      System.out.printf( "Math.tan( 0.0 ) = %f\n", Math.tan( 0.0 ) );
   } // end main
} // end class MathTest
```

```
Math.abs( 23.7 ) = 23.700000
Math.abs( 0.0 ) = 0.000000
Math.abs( -23.7 ) = 23.700000
Math.ceil( 9.2 ) = 10.000000
Math.ceil( -9.8 ) = -9.000000
Math.cos( 0.0 ) = 1.000000
Math.exp( 1.0 ) = 2.718282
Math.exp( 2.0 ) = 7.389056
Math.floor( 9.2 ) = 9.000000
Math.floor( -9.8 ) = -10.000000
Math.log( Math.E ) = 1.000000
Math.log( Math.E * Math.E ) = 2.000000
Math.max( 2.3, 12.7 ) = 12.700000
Math.max( -2.3, -12.7 ) = -2.300000
Math.min( 2.3, 12.7 ) = 2.300000
Math.min( -2.3, -12.7 ) = -12.700000
Math.pow( 2.0, 7.0 ) = 128.000000
Math.pow( 9.0, 0.5 ) = 3.000000
Math.sin( 0.0 ) = 0.000000
Math.sqrt( 900.0 ) = 30.000000
Math.tan( 0.0 ) = 0.000000
```

D.4
a) `double hypotenuse( double side1, double side2 )`
b) `int smallest( int x, int y, int z )`
c) `void instructions()`
d) `float intToFloat( int number )`

D.5
a) Error: Method `h` is declared within method `g`.
Correction: Move the declaration of `h` outside the declaration of `g`.
b) Error: The method is supposed to return an integer, but does not.
Correction: Delete the variable `result`, and place the statement

```
return x + y;
```

to the method, or add the following statement at the end of the method body:

```
return result;
```

c) Error: The semicolon after the right parenthesis of the parameter list is incorrect, and the parameter a should not be redeclared in the method.
Correction: Delete the semicolon after the right parenthesis of the parameter list, and delete the declaration `float a;`.

D.6 The following solution calculates the area of a circle, using the radius entered by the user:

```
// Exercise D.6: Radius.java
// Calculate the area of a circle.
import java.util.Scanner;

public class Circle
{
   // obtain radius from user and display area of circle
   public static void main( String[] args )
   {
      Scanner input = new Scanner( System.in );
      System.out.print( "Enter radius of circle: " );
      double radius = input.nextDouble();
      System.out.printf( "Area is %f\n", circleArea( radius ) );
   } // end method determineCircleArea

   // calculate and return circle area
   public static double circleArea( double radius )
   {
      double area = Math.PI * Math.pow( radius, 2 );
      return area;
   } // end method circleArea
} // end class Circle
```

```
Enter radius of circle: 5
Area is 78.571428
```

Exercises

D.7 What is the value of x after each of the following statements is executed?
a) `x = Math.abs( 7.5 );`
b) `x = Math.floor( 7.5 );`
c) `x = Math.abs( 0.0 );`
d) `x = Math.ceil( 0.0 );`
e) `x = Math.abs( -6.4 );`
f) `x = Math.ceil( -6.4 );`
g) `x = Math.ceil( -Math.abs( -8 + Math.floor( -5.5 ) ) );`

D.8 *(Parking Charges)* A parking garage charges a $2.00 minimum fee to park for up to three hours. The garage charges an additional $0.50 per hour for each hour *or part thereof* in excess of three hours. The maximum charge for any given 24-hour period is $10.00. Assume that no car parks for longer than 24 hours at a time. Write an application that calculates and displays the parking charges for each customer who parked in the garage yesterday. You should enter the hours parked for each customer. The program should display the charge for the current customer and should calculate and display the running total of yesterday's receipts. It should use the method `calculateCharges` to determine the charge for each customer.

D.9 *(Rounding Numbers)* `Math.floor` can be used to round values to the nearest integer—e.g.,

```
y = Math.floor( x + 0.5 );
```

will round the number `x` to the nearest integer and assign the result to `y`. Write an application that reads `double` values and uses the preceding statement to round each of the numbers to the nearest integer. For each number processed, display both the original number and the rounded number.

D.10 *(Rounding Numbers)* To round numbers to specific decimal places, use a statement like

```
y = Math.floor( x * 10 + 0.5 ) / 10;
```

which rounds `x` to the tenths position (i.e., the first position to the right of the decimal point), or

```
y = Math.floor( x * 100 + 0.5 ) / 100;
```

which rounds `x` to the hundredths position (i.e., the second position to the right of the decimal point). Write an application that defines four methods for rounding a number `x` in various ways:

a) `roundToInteger( number )`
b) `roundToTenths( number )`
c) `roundToHundredths( number )`
d) `roundToThousandths( number )`

For each value read, your program should display the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

D.11 Answer each of the following questions:

a) What is the difference between random and pseudorandom numbers?
b) Why is the `nextInt` method of class `Random` useful for simulating games of chance?
c) What are scaling factor and shifting values in random-number generation?
d) Why is computerized simulation of real-world situations a useful technique?

D.12 Write statements that assign random integers to the variable n in the following ranges:

a) $1 \le n \le 2$.
b) $1 \le n \le 100$.
c) $0 \le n \le 9$.
d) $1000 \le n \le 1112$.
e) $-1 \le n \le 1$.
f) $-3 \le n \le 11$.

D.13 Write statements that will display a random number from each of the following sets:

a) `8, 10, 12, 14, 16.`
b) `3, 5, 7, 9, 11.`
c) `5, 10, 15, 20, 25.`

D.14 *(Exponentiation)* Write a method `integerPower(base, exponent)` that returns the value of

$$base^{exponent}$$

For example, `integerPower(3, 4)` calculates 3^4 (or `3 * 3 * 3 * 3`). Assume that `exponent` is a positive, nonzero integer and that `base` is an integer. Use a `for` or `while` statement to control the calculation. Do not use any `Math` class methods. Incorporate this method into an application that reads integer values for `base` and `exponent` and performs the calculation with the `integerPower` method.

D.15 *(Multiples)* Write a method `isMultiple` that determines, for a pair of integers, whether the second integer is a multiple of the first. The method should take two integer arguments and return `true` if the second is a multiple of the first and `false` otherwise. [*Hint:* Use the remainder operator.] Incorporate this method into an application that inputs a series of pairs of integers (one pair at a time) and determines whether the second value in each pair is a multiple of the first.

D.16 *(Even or Odd)* Write a method `isEven` that uses the remainder operator (%) to determine whether an integer is even. The method should take an integer argument and return `true` if the integer is even and `false` otherwise. Incorporate this method into an application that inputs a sequence of integers (one at a time) and determines whether each is even or odd.

D.17 *(Circle Circumference)* Write an application that prompts the user for the radius of a circle and uses a method called `circleCircumference` to calculate the circumference of the circle.

D.18 *(Temperature Conversions)* Implement the following integer methods:

a) Method `celsius` returns the Celsius equivalent of a Fahrenheit temperature, using the calculation

```
celsius = 5.0 / 9.0 * ( fahrenheit - 32 );
```

b) Method `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature, using the calculation

```
fahrenheit = 9.0 / 5.0 * celsius + 32;
```

c) Use the methods from parts (a) and (b) to write an application that enables the user either to enter a Fahrenheit temperature and display the Celsius equivalent or to enter a Celsius temperature and display the Fahrenheit equivalent.

D.19 *(Find the Maximum)* Write a method `maximum3` that returns the largest of three floating-point numbers. Use the `Math.max` method to implement `maximum3`. Incorporate the method into an application that reads three values from the user, determines the largest value and displays the result.

D.20 *(Greatest Common Divisor)* The *greatest common divisor* (*GCD*) of two integers is the largest integer that evenly divides each of the two numbers. Write a method `gcd` that returns the greatest common divisor of two integers. [*Hint:* You might want to use Euclid's algorithm. You can find information about it at `en.wikipedia.org/wiki/Euclidean_algorithm`.] Incorporate the method into an application that reads two values from the user and displays the result.

D.21 *(Quality Points)* Write a method `qualityPoints` that inputs a student's average and returns `4` if it's 90–100, `3` if 80–89, `2` if 70–79, `1` if 60–69 and `0` if lower than 60. Incorporate the method into an application that reads a value from the user and displays the result.

D.22 *(Coin Tossing)* Write an application that simulates coin tossing. Let the program toss a coin each time the user chooses the "`Toss Coin`" menu option. Count the number of times each side of the coin appears. Display the results. The program should call a separate method `flip` that takes no arguments and returns a value from a `Coin` enum (`HEADS` and `TAILS`). [*Note:* If the program realistically simulates coin tossing, each side of the coin should appear approximately half the time.]

D.23 *(Guess the Number)* Write an application that plays "guess the number" as follows: Your program chooses the number to be guessed by selecting a random integer in the range 1 to 1000. The application displays the prompt `Guess a number between 1 and 1000`. The player inputs a first guess. If the player's guess is incorrect, your program should display `"Too high. Try again."` or `"Too low. Try again."` to help the player "zero in" on the correct answer. The program should prompt the user for the next guess. When the user enters the correct answer, display `"Congratulations. You` `guessed the number!"`, and allow the user to choose whether to play again. The guessing technique employed in this problem is similar to a binary search.

D.24 *(Craps Game Modification)* Modify the craps program of Fig. D.5 to allow wagering. Initialize variable `bankBalance` to 1000 dollars. Prompt the player to enter a `wager`. Check that `wager` is less than or equal to `bankBalance`, and if it's not, have the user reenter `wager` until a valid `wager` is entered. Then, run one game of craps. If the player wins, increase `bankBalance` by `wager` and display the new `bankBalance`. If the player loses, decrease `bankBalance` by `wager`, display the new `bankBalance`, check whether `bankBalance` has become zero and, if so, display the message `"Sorry. You busted!"` As the game progresses, display various messages to create some "chatter," such as `"Oh, you're going for broke, huh?"` or `"Aw c'mon, take a chance!"` or `"You're up big. Now's the time to cash in your chips!"`. Implement the "chatter" as a separate method that randomly chooses the string to display.

D.25 *(Computer-Assisted Instruction)* The use of computers in education is referred to as *computer-assisted instruction* (*CAI*). Write a program that will help an elementary school student learn multiplication. Use a `Random` object to produce two positive one-digit integers. The program should then prompt the user with a question, such as

```
How much is 6 times 7?
```

The student then inputs the answer. Next, the program checks the student's answer. If it's correct, display the message `"Very good!"` and ask another multiplication question. If the answer is wrong, display the message `"No. Please try again."` and let the student try the same question repeatedly until the student finally gets it right. A separate method should be used to generate each new question. This method should be called once when the application begins execution and each time the user answers the question correctly.

D.26 *(Computer-Assisted Instruction: Reducing Student Fatigue)* One problem in CAI environments is student fatigue. This can be reduced by varying the computer's responses to hold the student's attention. Modify the program of Exercise D.25 so that various comments are displayed for each answer as follows:

Possible responses to a correct answer:

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Possible responses to an incorrect answer:

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Use random-number generation to choose a number from 1 to 4 that will be used to select one of the four appropriate responses to each correct or incorrect answer. Use a `switch` statement to issue the responses.

D.27 *(Computer-Assisted Instruction: Varying the Types of Problems)* Modify the previous program to allow the user to pick a type of arithmetic problem to study. An option of `1` means addition problems only, `2` means subtraction problems only, `3` means multiplication problems only, `4` means division problems only and `5` means a random mixture of all these types.

E Arrays and ArrayLists

Objectives

In this appendix you'll learn:

- What arrays are.
- To use arrays to store data in and retrieve data from lists and tables of values.
- To declare arrays, initialize arrays and refer to individual elements of arrays.
- To iterate through arrays with the enhanced `for` statement.
- To pass arrays to methods.
- To declare and manipulate multidimensional arrays.
- To perform common array manipulations with the methods of class `Arrays`.
- To use class `ArrayList` to manipulate a dynamically resizable array-like data structure.

Outline

E.1 Introduction

This appendix introduces **data structures**—collections of related data items. **Arrays** are data structures consisting of related data items of the same type. Arrays make it convenient to process related groups of values. Arrays remain the same length once they're created, although an array variable may be reassigned such that it refers to a new array of a different length.

Although commonly used, arrays have limited capabilities. For instance, you must specify an array's size, and if at execution time you wish to modify it, you must do so manually by creating a new array. At the end of this appendix, we introduce one of Java's prebuilt data structures from the Java API's collection classes. These offer greater capabilities than traditional arrays. We focus on the `ArrayList` collection. `ArrayLists` are similar to arrays but provide additional functionality, such as **dynamic resizing**—they automatically increase their size at execution time to accommodate additional elements.

E.2 Arrays

An array is a group of variables (called **elements** or **components**) containing values that all have the same type. Arrays are *objects*, so they're considered reference types. As you'll soon see, what we typically think of as an array is actually a reference to an array object in memory. The *elements* of an array can be either primitive types or reference types (including arrays, as we'll see in Section E.9). To refer to a particular element in an array, we specify the name of the reference to the array and the *position number* of the element in the array. The position number of the element is called the element's **index** or **subscript**.

Figure E.1 shows a logical representation of an integer array called `c`. This array contains 12 elements. A program refers to any one of these elements with an **array-access expression** that includes the name of the array followed by the index of the particular element in **square brackets** (`[]`). The first element in every array has **index zero** and is sometimes called the **zeroth element.** Thus, the elements of array `c` are `c[0]`, `c[1]`, `c[2]` and so on. The highest index in array `c` is 11, which is 1 less than 12—the number of elements in the array. Array names follow the same conventions as other variable names.

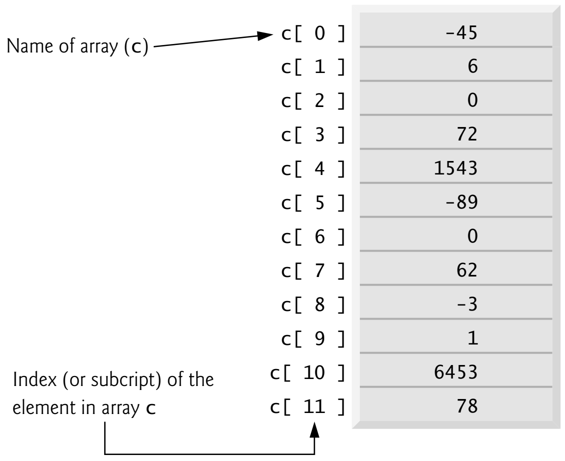


Fig. E.1 | A 12-element array.

An index must be a nonnegative integer. A program can use an expression as an index. For example, if we assume that variable `a` is `5` and variable `b` is `6`, then the statement

```
    c[ a + b ] += 2;
```

adds `2` to array element `c[11]`. An indexed array name is an array-access expression, which can be used on the left side of an assignment to place a new value into an array element.

Common Programming Error E.1
An index must be an `int` value or a value of a type that can be promoted to `int`—namely, `byte`, `short` or `char`, but not `long`; otherwise, a compilation error occurs.

Let's examine array `c` in Fig. E.1 more closely. The **name** of the array is `c`. Every array object knows its own length and stores it in a **`length` instance variable**. The expression `c.length` accesses array `c`'s `length` field to determine the length of the array. Even though the `length` instance variable of an array is `public`, it cannot be changed because it's a `final` variable. This array's 12 elements are referred to as `c[0]`, `c[1]`, `c[2]`, …, `c[11]`. The value of `c[0]` is `-45`, the value of `c[1]` is `6`, the value of `c[2]` is `0`, the value of `c[7]` is `62` and the value of `c[11]` is `78`. To calculate the sum of the values contained in the first three elements of array `c` and store the result in variable `sum`, we would write

```
    sum = c[ 0 ] + c[ 1 ] + c[ 2 ];
```

To divide the value of `c[6]` by `2` and assign the result to the variable `x`, we would write

```
    x = c[ 6 ] / 2;
```

E.3 Declaring and Creating Arrays

Array objects occupy space in memory. Like other objects, arrays are created with keyword `new`. To create an array object, you specify the type of the array elements and the number of elements as part of an **array-creation expression** that uses keyword `new`. Such an expression returns a reference that can be stored in an array variable. The following declaration and array-creation expression create an array object containing 12 `int` elements and store the array's reference in array variable `c`:

```
int[] c = new int[ 12 ];
```

This expression can be used to create the array shown in Fig. E.1. When an array is created, each element of the array receives a default value—zero for the numeric primitive-type elements, `false` for `boolean` elements and `null` for references. As you'll soon see, you can provide nondefault initial element values when you create an array.

Creating the array in Fig. E.1 can also be performed in two steps as follows:

```
int[] c; // declare the array variable
c = new int[ 12 ]; // create the array; assign to array variable
```

In the declaration, the square brackets following the type indicate that `c` is a variable that will refer to an array (i.e., the variable will store an array reference). In the assignment statement, the array variable `c` receives the reference to a new array of 12 `int` elements.

A program can create several arrays in a single declaration. The following declaration reserves 100 elements for `b` and 27 elements for `x`:

```
String[] b = new String[ 100 ], x = new String[ 27 ];
```

When the type of the array and the square brackets are combined at the beginning of the declaration, all the identifiers in the declaration are array variables. In this case, variables `b` and `x` refer to `String` arrays. For readability, we prefer to declare only one variable per declaration. The preceding declaration is equivalent to:

```
String[] b = new String[ 100 ]; // create array b
String[] x = new String[ 27 ]; // create array x
```

When only one variable is declared in each declaration, the square brackets can be placed either after the type or after the array variable name, as in:

```
String b[] = new String[ 100 ]; // create array b
String x[] = new String[ 27 ]; // create array x
```

Common Programming Error E.2

Declaring multiple array variables in a single declaration can lead to subtle errors. Consider the declaration `int[] a, b, c;`. If `a`, `b` and `c` should be declared as array variables, then this declaration is correct—placing square brackets directly following the type indicates that all *the identifiers in the declaration are array variables. However, if only `a` is intended to be an array variable, and `b` and `c` are intended to be individual `int` variables, then this declaration is incorrect—the declaration `int a[], b, c;` would achieve the desired result.*

A program can declare arrays of any type. Every element of a primitive-type array contains a value of the array's declared element type. Similarly, in an array of a reference type, every element is a reference to an object of the array's declared element type. For example, every element of an `int` array is an `int` value, and every element of a `String` array is a reference to a `String` object.

E.4 Examples Using Arrays

This section presents several examples that demonstrate declaring arrays, creating arrays, initializing arrays and manipulating array elements.

Creating and Initializing an Array

The application of Fig. E.2 uses keyword `new` to create an array of 10 `int` elements, which are initially zero (the default for `int` variables). Line 8 declares `array`—a reference capable of referring to an array of `int` elements. Line 10 creates the array object and assigns its reference to variable `array`. Line 12 outputs the column headings. The first column contains the index (0–9) of each array element, and the second column contains the default value (0) of each array element.

```
 1  // Fig. E.2: InitArray.java
 2  // Initializing the elements of an array to default values of zero.
 3
 4  public class InitArray
 5  {
 6     public static void main( String[] args )
 7     {
 8        int[] array; // declare array named array
 9
10        array = new int[ 10 ]; // create the array object
11
12        System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
13
14        // output each array element's value
15        for ( int counter = 0; counter < array.length; counter++ )
16           System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17     } // end main
18  } // end class InitArray
```

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

Fig. E.2 | Initializing the elements of an array to default values of zero.

The `for` statement in lines 15–16 outputs the index number (represented by `counter`) and the value of each array element (represented by `array[counter]`). The loop-control variable `counter` is initially `0`—index values start at 0, so using **zero-based counting** allows the loop to access every element of the array. The `for`'s loop-continuation condition uses the expression `array.length` (line 15) to determine the length of the array. In this example, the length of the array is 10, so the loop continues executing as long as the value of control variable `counter` is less than 10. The highest index value of a 10-element array is 9, so using the less-than operator in the loop-continuation condition guarantees that the loop does not attempt to access an element *beyond* the end of the array (i.e., during the final iteration of the loop, `counter` is `9`). We'll soon see what Java does when it encounters such an *out-of-range index* at execution time.

Using an Array Initializer

You can create an array and initialize its elements with an **array initializer**—a comma-separated list of expressions (called an **initializer list**) enclosed in braces. In this case, the array length is determined by the number of elements in the initializer list. For example,

```
      int[] n = { 10, 20, 30, 40, 50 };
```

creates a five-element array with index values 0–4. Element n[0] is initialized to 10, n[1] is initialized to 20, and so on. When the compiler encounters an array declaration that includes an initializer list, it counts the number of initializers in the list to determine the size of the array, then sets up the appropriate new operation "behind the scenes."

The application in Fig. E.3 initializes an integer array with 10 values (line 9) and displays the array in tabular format. The code for displaying the array elements (lines 14–15) is identical to that in Fig. E.2 (lines 15–16).

```
 1  // Fig. E.3: InitArray.java
 2  // Initializing the elements of an array with an array initializer.
 3
 4  public class InitArray
 5  {
 6     public static void main( String[] args )
 7     {
 8        // initializer list specifies the value for each element
 9        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11        System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
12
13        // output each array element's value
14        for ( int counter = 0; counter < array.length; counter++ )
15           System.out.printf( "%5d%8d\n", counter, array[ counter ] );
16     } // end main
17  } // end class InitArray
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Fig. E.3 | Initializing the elements of an array with an array initializer.

Calculating the Values to Store in an Array

The application in Fig. E.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (2, 4, 6, …, 20). Then the application displays the array in tabular format. The for statement at lines 12–13 calculates an array element's value by multiplying the current value of the control variable counter by 2, then adding 2.

```
 1  // Fig. E.4: InitArray.java
 2  // Calculating the values to be placed into the elements of an array.
 3
 4  public class InitArray
 5  {
 6     public static void main( String[] args )
 7     {
 8        final int ARRAY_LENGTH = 10; // declare constant
 9        int[] array = new int[ ARRAY_LENGTH ]; // create array
10
11        // calculate value for each array element
12        for ( int counter = 0; counter < array.length; counter++ )
13           array[ counter ] = 2 + 2 * counter;
14
15        System.out.printf( "%s%8s\n", "Index", "Value" ); // column headings
16
17        // output each array element's value
18        for ( int counter = 0; counter < array.length; counter++ )
19           System.out.printf( "%5d%8d\n", counter, array[ counter ] );
20     } // end main
21  } // end class InitArray
```

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Fig. E.4 | Calculating the values to be placed into the elements of an array.

Line 8 uses the modifier `final` to declare the constant variable `ARRAY_LENGTH` with the value 10. Constant variables must be initialized before they're used and cannot be modified thereafter. If you attempt to *modify* a `final` variable after it's initialized in its declaration, the compiler issues an error message like

```
cannot assign a value to final variable variableName
```

If an attempt is made to access the value of a `final` variable before it's initialized, the compiler issues an error message like

```
variable variableName might not have been initialized
```

Good Programming Practice E.1

Constant variables also are called ***named constants****. They often make programs more readable than programs that use literal values (e.g., 10)—a named constant such as* `ARRAY_LENGTH` *clearly indicates its purpose, whereas a literal value could have different meanings based on its context.*

Using Bar Charts to Display Array Data Graphically

Many programs present data to users in a graphical manner. For example, numeric values are often displayed as bars in a bar chart. In such a chart, longer bars represent proportionally larger numeric values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (*).

Professors often like to examine the distribution of grades on an exam. A professor might graph the number of grades in each of several categories to visualize the grade distribution. Suppose the grades on an exam were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. They include one grade of 100, two grades in the 90s, four grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Our next application (Fig. E.5) stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, `array[0]` indicates the number of grades in the range 0–9, `array[7]` the number of grades in the range 70–79 and `array[10]` the number of 100 grades.

```java
// Fig. E.5: BarChart.java
// Bar chart printing program.

public class BarChart
{
   public static void main( String[] args )
   {
      int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

      System.out.println( "Grade distribution:" );

      // for each array element, output a bar of the chart
      for ( int counter = 0; counter < array.length; counter++ )
      {
         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
         if ( counter == 10 )
            System.out.printf( "%5d: ", 100 );
         else
            System.out.printf( "%02d-%02d: ",
               counter * 10, counter * 10 + 9  );

         // print bar of asterisks
         for ( int stars = 0; stars < array[ counter ]; stars++ )
            System.out.print( "*" );

         System.out.println(); // start a new line of output
      } // end outer for
   } // end main
} // end class BarChart
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
```

Fig. E.5 | Bar chart printing program. (Part 1 of 2.)

```
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. E.5 | Bar chart printing program. (Part 2 of 2.)

The application reads the numbers from the array and graphs the information as a bar chart. It displays each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 16–20 output a grade range (e.g., `"70-79: "`) based on the current value of `counter`. When `counter` is `10`, line 17 outputs `100` with a field width of 5, followed by a colon and a space, to align the label `"100: "` with the other bar labels. The nested `for` statement (lines 23–24) outputs the bars. Note the loop-continuation condition at line 23 (`stars < array[counter]`). Each time the program reaches the inner `for`, the loop counts from `0` up to `array[counter]`, thus using a value in `array` to determine the number of asterisks to display. In this example, no students received a grade below 60, so `array[0]`–`array[5]` contain zeroes, and no asterisks are displayed next to the first six grade ranges. In line 19, the format specifier `%02d` indicates that an `int` value should be formatted as a field of two digits. The **`0` flag** in the format specifier displays a leading `0` for values with fewer digits than the field width (`2`).

Using the Elements of an Array as Counters

Sometimes, programs use counter variables to summarize data, such as the results of a survey. Figure E.6 uses the array `frequency` (line 10) to count the occurrences of each side of the die that's rolled 6,000,000 times. Line 14 uses the random value to determine which `frequency` element to increment during each iteration of the loop. The calculation in line 14 produces random numbers from 1 to 6, so the array `frequency` must be large enough to store six counters. However, we use a seven-element array in which we ignore `frequency[0]`—it's more logical to have the face value 1 increment `frequency[1]` than `frequency[0]`. Thus, each face value is used as an index for array `frequency`. In line 14, the calculation inside the square brackets evaluates first to determine which element of the array to increment, then the `++` operator adds one to that element. Lines 19–20 loop through array `frequency` to output the results.

```
 1  // Fig. E.7: RollDie.java
 2  // Die-rolling program using arrays instead of switch.
 3  import java.util.Random;
 4
 5  public class RollDie
 6  {
 7     public static void main( String[] args )
 8     {
 9        Random randomNumbers = new Random(); // random number generator
10        int[] frequency = new int[ 7 ]; // array of frequency counters
11
```

Fig. E.6 | Die-rolling program using arrays instead of `switch`. (Part 1 of 2.)

```
      // roll die 6,000,000 times; use die value as frequency index
      for ( int roll = 1; roll <= 6000000; roll++ )
         ++frequency[ 1 + randomNumbers.nextInt( 6 ) ];

      System.out.printf( "%s%10s\n", "Face", "Frequency" );

      // output each array element's value
      for ( int face = 1; face < frequency.length; face++ )
         System.out.printf( "%4d%10d\n", face, frequency[ face ] );
   } // end main
} // end class RollDie
```

```
Face Frequency
   1    999690
   2    999512
   3   1000575
   4    999815
   5    999781
   6   1000627
```

Fig. E.6 | Die-rolling program using arrays instead of `switch`. (Part 2 of 2.)

Using Arrays to Analyze Survey Results

Our next example uses arrays to summarize data collected in a survey. Consider the following problem statement:

> *Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an integer array and determine the frequency of each rating.*

This is a typical array-processing application (Fig. E.7). We wish to summarize the number of responses of each type (that is, 1–5). Array `responses` (lines 9–10) is a 20-element integer array containing the students' survey responses. The last value in the array is intentionally an incorrect response (`14`). When a Java program executes, array element indices are checked for validity—all indices must be greater than or equal to 0 and less than the length of the array. Any attempt to access an element outside that range of indices results in a runtime error that's known as an `ArrayIndexOutOfBoundsException`. At the end of this section, we'll discuss the invalid response value, demonstrate array **bounds checking** and introduce Java's exception-handling mechanism, which can be used to detect and handle an `ArrayIndexOutOfBoundsException`.

```
 1  // Fig. E.7: StudentPoll.java
 2  // Poll analysis program.
 3
 4  public class StudentPoll
 5  {
 6     public static void main( String[] args )
 7     {
 8        // student response array (more typically, input at runtime)
 9        int[] responses = { 1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
10           2, 3, 3, 2, 14 };
```

Fig. E.7 | Poll analysis program. (Part 1 of 2.)

```
11        int[] frequency = new int[ 6 ]; // array of frequency counters
12
13        // for each answer, select responses element and use that value
14        // as frequency index to determine element to increment
15        for ( int answer = 0; answer < responses.length; answer++ )
16        {
17           try
18           {
19              ++frequency[ responses[ answer ] ];
20           } // end try
21           catch ( ArrayIndexOutOfBoundsException e )
22           {
23              System.out.println( e );
24              System.out.printf( "   responses[%d] = %d\n\n",
25                 answer, responses[ answer ] );
26           } // end catch
27        } // end for
28
29        System.out.printf( "%s%10s\n", "Rating", "Frequency" );
30
31        // output each array element's value
32        for ( int rating = 1; rating < frequency.length; rating++ )
33           System.out.printf( "%6d%10d\n", rating, frequency[ rating ] );
34     } // end main
35  } // end class StudentPoll
```

```
java.lang.ArrayIndexOutOfBoundsException: 14
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

Fig. E.7 | Poll analysis program. (Part 2 of 2.)

The `frequency` *Array*

We use the *six-element* array `frequency` (line 11) to count the number of occurrences of each response. Each element is used as a counter for one of the possible types of survey responses—`frequency[1]` counts the number of students who rated the food as 1, `frequency[2]` counts the number of students who rated the food as 2, and so on.

Summarizing the Results

The `for` statement (lines 15–27) reads the responses from the array `responses` one at a time and increments one of the counters `frequency[1]` to `frequency[5]`; we ignore `frequency[0]` because the survey responses are limited to the range 1–5. The key statement in the loop appears in line 19. This statement increments the appropriate `frequency` counter as determined by the value of `responses[answer]`.

Let's step through the first few iterations of the `for` statement:

- When the counter `answer` is `0`, `responses[answer]` is the value of `responses[0]` (that is, `1`—see line 9). In this case, `frequency[responses[answer]]` is interpreted as `frequency[1]`, and the counter `frequency[1]` is incremented by one. To evaluate the expression, we begin with the value in the *innermost* set of brackets (`answer`, currently `0`). The value of `answer` is plugged into the expression, and the next set of brackets (`responses[answer]`) is evaluated. That value is used as the index for the `frequency` array to determine which counter to increment (in this case, `frequency[1]`).

- The next time through the loop `answer` is `1`, `responses[answer]` is the value of `responses[1]` (that is, `2`—see line 9), so `frequency[responses[answer]]` is interpreted as `frequency[2]`, causing `frequency[2]` to be incremented.
- When `answer` is `2`, `responses[answer]` is the value of `responses[2]` (that is, `5`—see line 9), so `frequency[responses[answer]]` is interpreted as `frequency[5]`, causing `frequency[5]` to be incremented, and so on.

Regardless of the number of responses processed in the survey, only a six-element array (in which we ignore element zero) is required to summarize the results, because all the correct response values are between 1 and 5, and the index values for a six-element array are 0–5. In the program's output, the `Frequency` column summarizes only 19 of the 20 values in the `responses` array—the last element of the array `responses` contains an incorrect response that was not counted.

Exception Handling: Processing the Incorrect Response

An **exception** indicates a problem that occurs while a program executes. The name "exception" suggests that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the problem represents the "exception to the rule." **Exception handling** enables you to create **fault-tolerant programs** that can resolve (or handle) exceptions. In many cases, this allows a program to continue executing as if no problems were encountered. For example, the `StudentPoll` application still displays results (Fig. E.7), even though one of the responses was out of range. More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate. When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it **throws** an exception—that is, an exception occurs.

The `try` Statement

To handle an exception, place any code that might throw an exception in a **`try` statement** (lines 17–26). The **`try` block** (lines 17–20) contains the code that might *throw* an exception, and the **`catch` block** (lines 21–26) contains the code that *handles* the exception if one occurs. You can have many `catch` blocks to handle different types of exceptions that might be thrown in the corresponding `try` block. When line 19 correctly increments an element of the `frequency` array, lines 21–26 are ignored. The braces that delimit the bodies of the `try` and `catch` blocks are required.

Executing the `catch` Block

When the program encounters the value `14` in the `responses` array, it attempts to add `1` to `frequency[14]`, which is *outside* the bounds of the array—the `frequency` array has only six elements. Because array bounds checking is performed at execution time, the JVM generates an exception—specifically line 19 throws an **`ArrayIndexOutOfBoundsException`** to notify the program of this problem. At this point the try block terminates and the catch block begins executing—if you declared any variables in the try block, they're now out of scope and are not accessible in the catch block.

The catch block declares a type (IndexOutOfRangeException) and an exception parameter (e). The catch block can handle exceptions of the specified type. Inside the catch block, you can use the parameter's identifier to interact with a caught exception object.

Error-Prevention Tip E.1

When writing code to access an array element, ensure that the array index remains greater than or equal to 0 and less than the length of the array. This helps prevent ArrayIndexOutOfBoundsException *in your program.*

***toString* Method of the Exception Parameter**

When lines 21–26 *catch* the exception, the program displays a message indicating the problem that occurred. Line 23 implicitly calls the exception object's toString method to get the error message that is stored in the exception object and display it. Once the message is displayed in this example, the exception is considered handled and the program continues with the next statement after the catch block's closing brace. In this example, the end of the for statement is reached (line 27), so the program continues with the increment of the control variable in line 15. We use exception handling again in Appendix F, and Appendix H presents a deeper look at exception handling.

E.5 Case Study: Card Shuffling and Dealing Simulation

The examples in the appendix thus far have used arrays containing elements of primitive types. Recall from Section E.2 that the elements of an array can be either primitive types or reference types. This section uses random-number generation and an array of reference-type elements, namely objects representing playing cards, to develop a class that simulates card shuffling and dealing. This class can then be used to implement applications that play specific card games.

We first develop class Card (Fig. E.8), which represents a playing card that has a face (e.g., "Ace", "Deuce", "Three", ..., "Jack", "Queen", "King") and a suit (e.g., "Hearts", "Diamonds", "Clubs", "Spades"). Next, we develop the DeckOfCards class (Fig. E.9), which creates a deck of 52 playing cards in which each element is a Card object. We then build a test application (Fig. E.10) that demonstrates class DeckOfCards's card-shuffling and dealing capabilities.

***Class* Card**

Class Card (Fig. E.8) contains two String instance variables—face and suit—that are used to store references to the face name and suit name for a specific Card. The constructor for the class (lines 10–14) receives two Strings that it uses to initialize face and suit. Method toString (lines 17–20) creates a String consisting of the face of the card, the String " of " and the suit of the card. Card's toString method can be invoked explicitly to obtain a string representation of a Card object (e.g., "Ace of Spades"). The toString method of an object is called *implicitly* when the object is used where a String is expected (e.g., when printf outputs the object as a String using the %s format specifier or when the object is concatenated to a `String` using the + operator). For this behavior to occur, `toString` must be declared with the header shown in Fig. E.8.

```java
// Fig. E.8: Card.java
// Card class represents a playing card.

public class Card
{
   private String face; // face of card ("Ace", "Deuce", ...)
   private String suit; // suit of card ("Hearts", "Diamonds", ...)

   // two-argument constructor initializes card's face and suit
   public Card( String cardFace, String cardSuit )
   {
      face = cardFace; // initialize face of card
      suit = cardSuit; // initialize suit of card
   } // end two-argument Card constructor

   // return String representation of Card
   public String toString()
   {
      return face + " of " + suit;
   } // end method toString
} // end class Card
```

Fig. E.8 | `Card` class represents a playing card.

Class `DeckOfCards`

Class `DeckOfCards` (Fig. E.9) declares as an instance variable a `Card` array named `deck` (line 7). An array of a reference type is declared like any other array. Class `DeckOfCards` also declares an integer instance variable `currentCard` (line 8) representing the next `Card` to be dealt from the `deck` array and a named constant `NUMBER_OF_CARDS` (line 9) indicating the number of `Cards` in the deck (52).

```java
 1  // Fig. E.9: DeckOfCards.java
 2  // DeckOfCards class represents a deck of playing cards.
 3  import java.util.Random;
 4
 5  public class DeckOfCards
 6  {
 7     private Card[] deck; // array of Card objects
 8     private int currentCard; // index of next Card to be dealt (0-51)
 9     private static final int NUMBER_OF_CARDS = 52; // constant # of Cards
10     // random number generator
11     private static final Random randomNumbers = new Random();
12
13     // constructor fills deck of Cards
14     public DeckOfCards()
15     {
```

Fig. E.9 | `DeckOfCards` class represents a deck of playing cards. (Part 1 of 2.)

```
16          String[] faces = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
17             "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
18          String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
19
20          deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
21          currentCard = 0; // set currentCard so first Card dealt is deck[ 0 ]
22
23          // populate deck with Card objects
24          for ( int count = 0; count < deck.length; count++ )
25             deck[ count ] =
26                new Card( faces[ count % 13 ], suits[ count / 13 ] );
27       } // end DeckOfCards constructor
28
29       // shuffle deck of Cards with one-pass algorithm
30       public void shuffle()
31       {
32          // after shuffling, dealing should start at deck[ 0 ] again
33          currentCard = 0; // reinitialize currentCard
34
35          // for each Card, pick another random Card (0-51) and swap them
36          for ( int first = 0; first < deck.length; first++ )
37          {
38             // select a random number between 0 and 51
39             int second = randomNumbers.nextInt( NUMBER_OF_CARDS );
40
41             // swap current Card with randomly selected Card
42             Card temp = deck[ first ];
43             deck[ first ] = deck[ second ];
44             deck[ second ] = temp;
45          } // end for
46       } // end method shuffle
47
48       // deal one Card
49       public Card dealCard()
50       {
51          // determine whether Cards remain to be dealt
52          if ( currentCard < deck.length )
53             return deck[ currentCard++ ]; // return current Card in array
54          else
55             return null; // return null to indicate that all Cards were dealt
56       } // end method dealCard
57    } // end class DeckOfCards
```

Fig. E.9 | `DeckOfCards` class represents a deck of playing cards. (Part 2 of 2.)

***`DeckOfCards`* Constructor**

The class's constructor instantiates array `deck` (line 20) with `NUMBER_OF_CARDS` (52) elements that are all `null` by default. Lines 24–26 fill the `deck` with `Card`s. The loop initializes control variable `count` to `0` and loops while `count` is less than `deck.length`, causing `count` to take on each integer value from 0 to 51 (the indices of array `deck`). Each `Card` is instantiated and initialized with a `String` from the `faces` array (which contains `"Ace"` through `"King"`) and a `String` from the `suits` array (which contains `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`). The calculation `count % 13` always results in a value from 0 to 12 (the 13 indices of the `faces` array in lines 16–17), and the calculation `count / 13` always results in a value from 0 to 3 (the four indices of the `suits` array in line 18). When the `deck` array is initialized, it contains the `Card`s with faces `"Ace"` through `"King"` in order for each suit (`"Hearts"` then `"Diamonds"` then `"Clubs"` then `"Spades"`).

`DeckOfCards` *Method* `shuffle`

Method `shuffle` (lines 30–46) shuffles the `Card`s in the deck. The method loops through all 52 `Card`s. For each `Card`, a number between 0 and 51 is picked randomly to select another `Card`, then the current `Card` and the randomly selected `Card` are swapped in the array. This exchange is performed by the assignments in lines 42–44. The extra variable `temp` temporarily stores one of the two `Card` objects being swapped. The swap cannot be performed with only the two statements

```
deck[ first ] = deck[ second ];
deck[ second ] = deck[ first ];
```

If `deck[first]` is the `"Ace"` of `"Spades"` and `deck[second]` is the `"Queen"` of `"Hearts"`, after the first assignment, both array elements contain the `"Queen"` of `"Hearts"` and the `"Ace"` of `"Spades"` is lost—hence, the extra variable `temp` is needed. After the `for` loop terminates, the `Card` objects are randomly ordered. A total of only 52 swaps are made in a single pass of the entire array, and the array of `Card` objects is shuffled!

[*Note:* It's recommended that you use a so-called unbiased shuffling algorithm for real card games. Such an algorithm ensures that all possible shuffled card sequences are equally likely to occur. A popular unbiased shuffling algorithm is the Fisher-Yates algorithm.]

`DeckOfCards` *Method* `dealCard`

Method `dealCard` (lines 49–56) deals one `Card` in the array. Recall that `currentCard` indicates the index of the next `Card` to be dealt (i.e., the `Card` at the top of the deck). Thus, line 52 compares `currentCard` to the array's length. If the `deck` is not empty (i.e., `currentCard` is less than 52), line 53 returns the "top" `Card` and postincrements `currentCard` to prepare for the next call to `dealCard`—otherwise, `null` is returned.

Shuffling and Dealing Cards

Figure E.10 demonstrates class `DeckOfCards` (Fig. E.9). Line 9 creates a `DeckOfCards` object named `myDeckOfCards`. The `DeckOfCards` constructor creates the deck with the 52 `Card` objects in order by suit and face. Line 10 invokes `myDeckOfCards`'s `shuffle` method to rearrange the `Card` objects. Lines 13–20 deal all 52 `Card`s and print them in four columns of 13 `Card`s each. Line 16 deals one `Card` object by invoking `myDeckOfCards`'s `dealCard` method, then displays the `Card` left justified in a field of 19 characters. When a `Card` is output as a `String`, the `Card`'s `toString` method (lines 17–20 of Fig. E.8) is implicitly invoked. Lines 18–19 (Fig. E.10) start a new line after every four `Card`s.

```
1  // Fig. E.10: DeckOfCardsTest.java
2  // Card shuffling and dealing.
3
4  public class DeckOfCardsTest
5  {
```

Fig. E.10 | Card shuffling and dealing. (Part 1 of 2.)

```
   // execute application
   public static void main( String[] args )
   {
      DeckOfCards myDeckOfCards = new DeckOfCards();
      myDeckOfCards.shuffle(); // place Cards in random order

      // print all 52 Cards in the order in which they are dealt
      for ( int i = 1; i <= 52; i++ )
      {
         // deal and display a Card
         System.out.printf( "%-19s", myDeckOfCards.dealCard() );

         if ( i % 4 == 0 ) // output a newline after every fourth card
            System.out.println();
      } // end for
   } // end main
} // end class DeckOfCardsTest
```

```
Six of Spades      Eight of Spades    Six of Clubs       Nine of Hearts
Queen of Hearts    Seven of Clubs     Nine of Spades     King of Hearts
Three of Diamonds  Deuce of Clubs     Ace of Hearts      Ten of Spades
Four of Spades     Ace of Clubs       Seven of Diamonds  Four of Hearts
Three of Clubs     Deuce of Hearts    Five of Spades     Jack of Diamonds
King of Clubs      Ten of Hearts      Three of Hearts    Six of Diamonds
Queen of Clubs     Eight of Diamonds  Deuce of Diamonds  Ten of Diamonds
Three of Spades    King of Diamonds   Nine of Clubs      Six of Hearts
Ace of Spades      Four of Diamonds   Seven of Hearts    Eight of Clubs
Deuce of Spades    Eight of Hearts    Five of Hearts     Queen of Spades
Jack of Hearts     Seven of Spades    Four of Clubs      Nine of Diamonds
Ace of Diamonds    Queen of Diamonds  Five of Clubs      King of Spades
Five of Diamonds   Ten of Clubs       Jack of Spades     Jack of Clubs
```

Fig. E.10 | Card shuffling and dealing. (Part 2 of 2.)

E.6 Enhanced for Statement

The **enhanced `for` statement** iterates through the elements of an array *without* using a counter, thus avoiding the possibility of "stepping outside" the array. We show how to use the enhanced `for` statement with the Java API's prebuilt data structures (called collections) in Section E.12. The syntax of an enhanced `for` statement is:

```
for ( parameter : arrayName )
   statement
```

where *parameter* has a type and an identifier (e.g., `int number`), and *arrayName* is the array through which to iterate. The type of the parameter must be consistent with the type of the elements in the array. As the next example illustrates, the identifier represents successive element values in the array on successive iterations of the loop.

Figure E.11 uses the enhanced `for` statement (lines 12–13) to sum the integers in an array of student grades. The enhanced `for`'s parameter is of type `int`, because `array` contains `int` values—the loop selects one `int` value from the array during each iteration. The enhanced `for` statement iterates through successive values in the array one by one. The statement's header can be read as "for each iteration, assign the next element of `array` to int variable number, then execute the following statement." Thus, for each iteration, identifier number represents an int value in array. Lines 12–13 are equivalent to the following counter-controlled repetition statement, except that counter cannot be accessed in the body of the enhanced for statement:

```
for ( int counter = 0; counter < array.length; counter++ )
   total += array[ counter ];
```

```
 1  // Fig. E.11: EnhancedForTest.java
 2  // Using the enhanced for statement to total integers in an array.
 3
 4  public class EnhancedForTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
 9        int total = 0;
10
11        // add each element's value to total
12        for ( int number : array )
13           total += number;
14
15        System.out.printf( "Total of array elements: %d\n", total );
16     } // end main
17  } // end class EnhancedForTest
```

```
Total of array elements: 849
```

Fig. E.11 | Using the enhanced for statement to total integers in an array.

The enhanced for statement simplifies the code for iterating through an array. Note, however, that *the enhanced for statement can be used only to obtain array elements—it cannot be used to modify elements*. If your program needs to modify elements, use the traditional counter-controlled for statement.

The enhanced for statement can be used in place of the counter-controlled for statement whenever code looping through an array does *not* require access to the counter indicating the index of the current array element. For example, totaling the integers in an array requires access only to the element values—the index of each element is irrelevant. However, if a program must use a counter for some reason other than simply to loop through an array (e.g., to print an index number next to each array element value, as in the examples earlier in this appendix), use the counter-controlled for statement.

E.7 Passing Arrays to Methods

This section demonstrates how to pass arrays and individual array elements as arguments to methods. To pass an array argument to a method, specify the name of the array without any brackets. For example, if array hourlyTemperatures is declared as

```
double[] hourlyTemperatures = new double[ 24 ];
```

then the method call

```
    modifyArray( hourlyTemperatures );
```

passes the reference of array `hourlyTemperatures` to method `modifyArray`. Every array object "knows" its own length (via its `length` field). Thus, when we pass an array object's reference into a method, we need not pass the array length as an additional argument.

For a method to receive an array reference through a method call, the method's parameter list must specify an array parameter. For example, the method header for method `modifyArray` might be written as

```
    void modifyArray( double[] b )
```

indicating that `modifyArray` receives the reference of a `double` array in parameter `b`. The method call passes array `hourlyTemperature`'s reference, so when the called method uses the array variable `b`, it *refers to* the same array object as `hourlyTemperatures` in the caller.

When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a *copy* of the reference. However, when an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element's *value*. Such primitive values are called **scalars** or **scalar quantities**. To pass an individual array element to a method, use the indexed name of the array element as an argument in the method call.

Figure E.12 demonstrates the difference between passing an entire array and passing a primitive-type array element to a method. Notice that `main` invokes `static` methods `modifyArray` (line 19) and `modifyElement` (line 30) directly. Recall from Section D.4 that a `static` method of a class can invoke other `static` methods of the same class directly.

```
 1  // Fig. E.12: PassArray.java
 2  // Passing arrays and individual array elements to methods.
 3
 4  public class PassArray
 5  {
 6     // main creates array and calls modifyArray and modifyElement
 7     public static void main( String[] args )
 8     {
 9        int[] array = { 1, 2, 3, 4, 5 };
10
11        System.out.println(
12           "Effects of passing reference to entire array:\n" +
13           "The values of the original array are:" );
14
15        // output original array elements
16        for ( int value : array )
17           System.out.printf( "   %d", value );
18
19        modifyArray( array ); // pass array reference
20        System.out.println( "\n\nThe values of the modified array are:" );
21
22        // output modified array elements
23        for ( int value : array )
24           System.out.printf( "   %d", value );
25
```

Fig. E.12 | Passing arrays and individual array elements to methods. (Part 1 of 2.)

```
26          System.out.printf(
27             "\n\nEffects of passing array element value:\n" +
28             "array[3] before modifyElement: %d\n", array[ 3 ] );
29
30          modifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
31          System.out.printf(
32             "array[3] after modifyElement: %d\n", array[ 3 ] );
33       } // end main
34
35       // multiply each element of an array by 2
36       public static void modifyArray( int[] array2 )
37       {
38          for ( int counter = 0; counter < array2.length; counter++ )
39             array2[ counter ] *= 2;
40       } // end method modifyArray
41
42       // multiply argument by 2
43       public static void modifyElement( int element )
44       {
45          element *= 2;
46          System.out.printf(
47             "Value of element in modifyElement: %d\n", element );
48       } // end method modifyElement
49    } // end class PassArray
```

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

Fig. E.12 | Passing arrays and individual array elements to methods. (Part 2 of 2.)

The enhanced `for` statement at lines 16–17 outputs the five `int` elements of `array`. Line 19 invokes method `modifyArray`, passing `array` as an argument. Method `modifyArray` (lines 36–40) receives a copy of `array`'s reference and uses the reference to multiply each of `array`'s elements by 2. To prove that `array`'s elements were modified, lines 23–24 output the five elements of `array` again. As the output shows, method `modifyArray` doubled the value of each element. We could not use the enhanced `for` statement in lines 38–39 because we're modifying the array's elements.

Figure E.12 next demonstrates that when a copy of an individual primitive-type array element is passed to a method, modifying the *copy* in the called method does *not* affect the original value of that element in the calling method's array. Lines 26–28 output the value of `array[3]` *before* invoking method `modifyElement`. Remember that the value of this element is now 8 after it was modified in the call to `modifyArray`. Line 30 calls method `mod-`

`ifyElement` and passes `array[3]` as an argument. Remember that `array[3]` is actually one `int` value (8) in `array`. Therefore, the program passes a copy of the value of `array[3]`. Method `modifyElement` (lines 43–48) multiplies the value received as an argument by 2, stores the result in its parameter `element`, then outputs the value of `element` (16). Since method parameters, like local variables, cease to exist when the method in which they're declared completes execution, the method parameter `element` is destroyed when method `modifyElement` terminates. When the program returns control to `main`, lines 31–32 output the *unmodified* value of `array[3]` (i.e., 8).

Notes on Passing Arguments to Methods

The preceding example demonstrated how arrays and primitive-type array elements are passed as arguments to methods. We now take a closer look at how arguments in general are passed to methods. Two ways to pass arguments in method calls in many programming languages are **pass-by-value** and **pass-by-reference** (also called **call-by-value** and **call-by-reference**). When an argument is passed by value, a copy of the argument's *value* is passed to the called method. The called method works exclusively with the copy. Changes to the called method's copy do *not* affect the original variable's value in the caller.

When an argument is passed by reference, the called method can access the argument's value in the caller directly and modify that data, if necessary. Pass-by-reference improves performance by eliminating the need to copy possibly large amounts of data.

Unlike some other languages, Java does *not* allow you to choose pass-by-value or pass-by-reference—*all arguments are passed by value*. A method call can pass two types of values to a method—copies of primitive values (e.g., values of type `int` and `double`) and copies of references to objects. Objects themselves cannot be passed to methods. When a method modifies a primitive-type parameter, changes to the parameter have no effect on the original argument value in the calling method. For example, when line 30 in `main` of Fig. E.12 passes `array[3]` to method `modifyElement`, the statement in line 45 that doubles the value of parameter `element` has *no* effect on the value of `array[3]` in `main`. This is also true for reference-type parameters. If you modify a reference-type parameter so that it refers to another object, only the parameter refers to the new object—the reference stored in the caller's variable still refers to the original object.

Although an object's reference is passed by value, a method can still interact with the referenced object by calling its `public` methods using the copy of the object's reference. Since the reference stored in the parameter is a copy of the reference that was passed as an argument, the parameter in the called method and the argument in the calling method refer to the same object in memory. For example, in Fig. E.12, both parameter `array2` in method `modifyArray` and variable `array` in `main` refer to the *same* array object in memory. Any changes made using the parameter `array2` are carried out on the object that `array` references in the calling method. In Fig. E.12, the changes made in `modifyArray` using `array2` affect the contents of the array object referenced by `array` in `main`. Thus, with a reference to an object, the called method *can* manipulate the caller's object directly.

Performance Tip E.1

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would waste time and consume considerable storage for the copies of the arrays.

E.8 Case Study: Class GradeBook Using an Array to Store Grades

Previous versions of class `GradeBook` process a set of grades entered by the user, but do not maintain the individual grade values in instance variables of the class. Thus, repeat calculations require the user to reenter the same grades. One way to solve this problem would be to store each grade entered in an individual instance of the class. For example, we could create instance variables `grade1`, `grade2`, …, `grade10` in class `GradeBook` to store 10 student grades. But this would make the code to total the grades and determine the class average cumbersome, and the class would not be able to process any more than 10 grades at a time. We solve this problem by storing grades in an array.

Storing Student Grades in an Array in Class GradeBook

Class `GradeBook` (Fig. E.13) uses an array of `int`s to store several students' grades on a single exam. This eliminates the need to repeatedly input the same set of grades. Array `grades` is declared as an instance variable (line 7), so each `GradeBook` object maintains its own set of grades. The constructor (lines 10–14) has two parameters—the name of the course and an array of grades. When an application (e.g., class `GradeBookTest` in Fig. E.14) creates a `GradeBook` object, the application passes an existing `int` array to the constructor, which assigns the array's reference to instance variable `grades` (line 13). The `grades` array's size is determined by the length of the array that's passed to the constructor. Thus, a `GradeBook` object can process a variable number of grades. The grade values in the passed array could have been input from a user or read from a file on disk. In our test application, we initialize an array with grade values (Fig. E.14, line 10). Once the grades are stored in instance variable `grades` of class `GradeBook`, all the class's methods can access the elements of `grades` *as often as needed* to perform various calculations.

Method `processGrades` (lines 37–51) contains a series of method calls that output a report summarizing the grades. Line 40 calls method `outputGrades` to print the contents of the array `grades`. Lines 134–136 in method `outputGrades` use a `for` statement to output the students' grades. A counter-controlled `for` *must* be used in this case, because lines 135–136 use counter variable `student`'s value to output each grade next to a particular student number (see output in Fig. E.14). Although array indices start at 0, a professor would typically number students starting at 1. Thus, lines 135–136 output `student + 1` as the student number to produce grade labels `"Student 1: "`, `"Student 2: "`, and so on.

```
 1  // Fig. E.13: GradeBook.java
 2  // GradeBook class using an array to store test grades.
 3
 4  public class GradeBook
 5  {
 6     private String courseName; // name of course this GradeBook represents
 7     private int[] grades; // array of student grades
 8
 9     // two-argument constructor initializes courseName and grades array
10     public GradeBook( String name, int[] gradesArray )
11     {
```

Fig. E.13 | `GradeBook` class using an array to store test grades. (Part 1 of 4.)

```java
      courseName = name; // initialize courseName
      grades = gradesArray; // store grades
   } // end two-argument GradeBook constructor

   // method to set the course name
   public void setCourseName( String name )
   {
      courseName = name; // store the course name
   } // end method setCourseName

   // method to retrieve the course name
   public String getCourseName()
   {
      return courseName;
   } // end method getCourseName

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // perform various operations on the data
   public void processGrades()
   {
      // output grades array
      outputGrades();

      // call method getAverage to calculate the average grade
      System.out.printf( "\nClass average is %.2f\n", getAverage() );

      // call methods getMinimum and getMaximum
      System.out.printf( "Lowest grade is %d\nHighest grade is %d\n\n",
         getMinimum(), getMaximum() );

      // call outputBarChart to print grade distribution chart
      outputBarChart();
   } // end method processGrades

   // find minimum grade
   public int getMinimum()
   {
      int lowGrade = grades[ 0 ]; // assume grades[ 0 ] is smallest

      // loop through grades array
      for ( int grade : grades )
      {
         // if grade lower than lowGrade, assign it to lowGrade
         if ( grade < lowGrade )
            lowGrade = grade; // new lowest grade
      } // end for
```

Fig. E.13 | GradeBook class using an array to store test grades. (Part 2 of 4.)

```java

      return lowGrade; // return lowest grade
   } // end method getMinimum

   // find maximum grade
   public int getMaximum()
   {
      int highGrade = grades[ 0 ]; // assume grades[ 0 ] is largest

      // loop through grades array
      for ( int grade : grades )
      {
         // if grade greater than highGrade, assign it to highGrade
         if ( grade > highGrade )
            highGrade = grade; // new highest grade
      } // end for

      return highGrade; // return highest grade
   } // end method getMaximum

   // determine average grade for test
   public double getAverage()
   {
      int total = 0; // initialize total

      // sum grades for one student
      for ( int grade : grades )
         total += grade;

      // return average of grades
      return (double) total / grades.length;
   } // end method getAverage

   // output bar chart displaying grade distribution
   public void outputBarChart()
   {
      System.out.println( "Grade distribution:" );

      // stores frequency of grades in each range of 10 grades
      int[] frequency = new int[ 11 ];

      // for each grade, increment the appropriate frequency
      for ( int grade : grades )
         ++frequency[ grade / 10 ];

      // for each grade frequency, print bar in chart
      for ( int count = 0; count < frequency.length; count++ )
      {
         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
         if ( count == 10 )
            System.out.printf( "%5d: ", 100 );
```

Fig. E.13 | GradeBook class using an array to store test grades. (Part 3 of 4.)

```
116           else
117              System.out.printf( "%02d-%02d: ",
118                 count * 10, count * 10 + 9  );
119
120           // print bar of asterisks
121           for ( int stars = 0; stars < frequency[ count ]; stars++ )
122              System.out.print( "*" );
123
124           System.out.println(); // start a new line of output
125        } // end outer for
126     } // end method outputBarChart
127
128     // output the contents of the grades array
129     public void outputGrades()
130     {
131        System.out.println( "The grades are:\n" );
132
133        // output each student's grade
134        for ( int student = 0; student < grades.length; student++ )
135           System.out.printf( "Student %2d: %3d\n",
136              student + 1, grades[ student ] );
137     } // end method outputGrades
138  } // end class GradeBook
```

Fig. E.13 | GradeBook class using an array to store test grades. (Part 4 of 4.)

Method processGrades next calls method getAverage (line 43) to obtain the average of the grades in the array. Method getAverage (lines 86–96) uses an enhanced for statement to total the values in array grades before calculating the average. The parameter in the enhanced for's header (e.g., int grade) indicates that for each iteration, the int variable grade takes on a value in the array grades. The averaging calculation in line 95 uses grades.length to determine the number of grades being averaged.

Lines 46–47 in method processGrades call methods getMinimum and getMaximum to determine the lowest and highest grades of any student on the exam, respectively. Each of these methods uses an enhanced for statement to loop through array grades. Lines 59–64 in method getMinimum loop through the array. Lines 62–63 compare each grade to lowGrade; if a grade is less than lowGrade, lowGrade is set to that grade. When line 66 executes, lowGrade contains the lowest grade in the array. Method getMaximum (lines 70–83) works similarly to method getMinimum.

Finally, line 50 in method processGrades calls method outputBarChart to print a distribution chart of the grade data using a technique similar to that in Fig. E.5. In that example, we manually calculated the number of grades in each category (i.e., 0–9, 10–19, …, 90–99 and 100) by simply looking at a set of grades. In this example, lines 107–108 use a technique similar to that in Figs. E.6 and 7.8 to calculate the frequency of grades in each category. Line 104 declares and creates array frequency of 11 ints to store the frequency of grades in each grade category. For each grade in array grades, lines 107–108 increment the appropriate element of the frequency array. To determine which element to increment, line 108 divides the current grade by 10 using integer division. For example, if grade is 85, line 108 increments frequency[8] to update the count of grades in the range 80–89. Lines 111–125 next print the bar chart (see Fig. E.14) based on the values in array `frequency`. Like lines 23–24 of Fig. E.5, lines 121–122 of Fig. E.13 use a value in array `frequency` to determine the number of asterisks to display in each bar.

Class `GradeBookTest` *That Demonstrates Class* `GradeBook`

The application of Fig. E.14 creates an object of class `GradeBook` (Fig. E.13) using the `int` array `gradesArray` (declared and initialized in line 10 of Fig. E.14). Lines 12–13 pass a course name and `gradesArray` to the `GradeBook` constructor. Line 14 displays a welcome message, and line 15 invokes the `GradeBook` object's `processGrades` method. The output summarizes the 10 grades in `myGradeBook`.

Software Engineering Observation E.1

A test harness (or test application) is responsible for creating an object of the class being tested and providing it with data. This data could come from any of several sources. Test data can be placed directly into an array with an array initializer, it can come from the user at the keyboard, it can come from a file, or it can come from a network. After passing this data to the class's constructor to instantiate the object, the test harness should call upon the object to test its methods and manipulate its data. Gathering data in the test harness like this allows the class to manipulate data from several sources.

```java
 1  // Fig. E.14: GradeBookTest.java
 2  // GradeBookTest creates a GradeBook object using an array of grades,
 3  // then invokes method processGrades to analyze them.
 4  public class GradeBookTest
 5  {
 6     // main method begins program execution
 7     public static void main( String[] args )
 8     {
 9        // array of student grades
10        int[] gradesArray = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12        GradeBook myGradeBook = new GradeBook(
13           "CS101 Introduction to Java Programming", gradesArray );
14        myGradeBook.displayMessage();
15        myGradeBook.processGrades();
16     } // end main
17  } // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
```

Fig. E.14 | `GradeBookTest` creates a `GradeBook` object using an array of grades, then invokes method `processGrades` to analyze them. (Part 1 of 2.)

```
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. E.14 | `GradeBookTest` creates a `GradeBook` object using an array of grades, then invokes method `processGrades` to analyze them. (Part 2 of 2.)

E.9 Multidimensional Arrays

Multidimensional arrays with two dimensions are often used to represent *tables* of values consisting of information arranged in *rows* and *columns*. To identify a particular table element, we must specify two indices. *By convention*, the first identifies the element's row and the second its column. Arrays that require two indices to identify a particular element are called **two-dimensional arrays**. (Multidimensional arrays can have more than two dimensions.) Java does not support multidimensional arrays directly, but it does allow you to specify one-dimensional arrays whose elements are also one-dimensional arrays, thus achieving the same effect. Figure E.15 illustrates a two-dimensional array named `a` that contains three rows and four columns (i.e., a three-by-four array). In general, an array with *m* rows and *n* columns is called an ***m*-by-*n* array**.

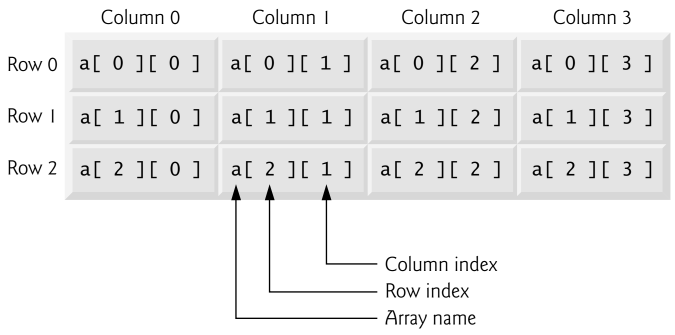


Fig. E.15 | Two-dimensional array with three rows and four columns.

Every element in array a is identified in Fig. E.15 by an *array-access expression* of the form a[*row*][*column*]; a is the name of the array, and *row* and *column* are the indices that uniquely identify each element in array a by row and column number. The names of the elements in *row* 0 all have a first index of 0, and the names of the elements in *column* 3 all have a second index of 3.

Arrays of One-Dimensional Arrays

Like one-dimensional arrays, multidimensional arrays can be initialized with array initializers in declarations. A two-dimensional array b with two rows and two columns could be declared and initialized with **nested array initializers** as follows:

```
    int[][] b = { { 1, 2 }, { 3, 4 } };
```

The initial values are grouped by row in braces. So 1 and 2 initialize b[0][0] and b[0][1], respectively, and 3 and 4 initialize b[1][0] and b[1][1], respectively. The compiler counts the number of nested array initializers (represented by sets of braces within the outer braces) to determine the number of rows in array b. The compiler counts the initializer values in the nested array initializer for a row to determine the number of columns in that row. As we'll see momentarily, this means that *rows can have different lengths*.

Multidimensional arrays are maintained as arrays of one-dimensional arrays. Therefore array b in the preceding declaration is actually composed of two separate one-dimensional arrays—one containing the values in the first nested initializer list { 1, 2 } and one containing the values in the second nested initializer list { 3, 4 }. Thus, array b itself is an array of two elements, each a one-dimensional array of int values.

Two-Dimensional Arrays with Rows of Different Lengths

The manner in which multidimensional arrays are represented makes them quite flexible. In fact, the lengths of the rows in array b are *not* required to be the same. For example,

```
    int[][] b = { { 1, 2 }, { 3, 4, 5 } };
```

creates integer array b with two elements (determined by the number of nested array initializers) that represent the rows of the two-dimensional array. Each element of b is a reference to a one-dimensional array of int variables. The int array for row 0 is a one-dimensional array with two elements (1 and 2), and the int array for row 1 is a one-dimensional array with three elements (3, 4 and 5).

Creating Two-Dimensional Arrays with Array-Creation Expressions

A multidimensional array with the same number of columns in every row can be created with an array-creation expression. For example, the following lines declare array b and assign it a reference to a three-by-four array:

```
    int[][] b = new int[ 3 ][ 4 ];
```

In this case, we use the literal values 3 and 4 to specify the number of rows and number of columns, respectively, but this is not required. Programs can also use variables to specify array dimensions, because *new creates arrays at execution time—not at compile time*. As with one-dimensional arrays, the elements of a multidimensional array are initialized when the array object is created.

A multidimensional array in which each row has a different number of columns can be created as follows:

```
int[][] b = new int[ 2 ][ ];   // create 2 rows
b[ 0 ] = new int[ 5 ]; // create 5 columns for row 0
b[ 1 ] = new int[ 3 ]; // create 3 columns for row 1
```

The preceding statements create a two-dimensional array with two rows. Row `0` has five columns, and row `1` has three columns.

Two-Dimensional Array Example: Displaying Element Values

Figure E.16 demonstrates initializing two-dimensional arrays with array initializers and using nested `for` loops to **traverse** the arrays (i.e., manipulate every element of each array). Class `InitArray`'s `main` declares two arrays. The declaration of `array1` (line 9) uses nested array initializers of the *same* length to initialize the first row to the values 1, 2 and 3, and the second row to the values 4, 5 and 6. The declaration of `array2` (line 10) uses nested initializers of *different* lengths. In this case, the first row is initialized to two elements with the values 1 and 2, respectively. The second row is initialized to one element with the value 3. The third row is initialized to three elements with the values 4, 5 and 6, respectively.

```
 1  // Fig. E.16: InitArray.java
 2  // Initializing two-dimensional arrays.
 3
 4  public class InitArray
 5  {
 6     // create and output two-dimensional arrays
 7     public static void main( String[] args )
 8     {
 9        int[][] array1 = { { 1, 2, 3 }, { 4, 5, 6 } };
10        int[][] array2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12        System.out.println( "Values in array1 by row are" );
13        outputArray( array1 ); // displays array1 by row
14
15        System.out.println( "\nValues in array2 by row are" );
16        outputArray( array2 ); // displays array2 by row
17     } // end main
18
19     // output rows and columns of a two-dimensional array
20     public static void outputArray( int[][] array )
21     {
22        // loop through array's rows
23        for ( int row = 0; row < array.length; row++ )
24        {
25           // loop through columns of current row
26           for ( int column = 0; column < array[ row ].length; column++ )
27              System.out.printf( "%d  ", array[ row ][ column ] );
28
29           System.out.println(); // start new line of output
30        } // end outer for
31     } // end method outputArray
32  } // end class InitArray
```

Fig. E.16 | Initializing two-dimensional arrays. (Part 1 of 2.)

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Fig. E.16 | Initializing two-dimensional arrays. (Part 2 of 2.)

Lines 13 and 16 call method `outputArray` (lines 20–31) to output the elements of `array1` and `array2`, respectively. Method `outputArray`'s parameter—`int[][] array`—indicates that the method receives a two-dimensional array. The `for` statement (lines 23–30) outputs the rows of a two-dimensional array. In the loop-continuation condition of the outer `for` statement, the expression `array.length` determines the number of rows in the array. In the inner `for` statement, the expression `array[row].length` determines the number of columns in the current row of the array. The inner `for` statement's condition enables the loop to determine the exact number of columns in each row.

Common Multidimensional-Array Manipulations Performed with `for` Statements

Many common array manipulations use `for` statements. As an example, the following `for` statement sets all the elements in row `2` of array `a` in Fig. E.15 to zero:

```
for ( int column = 0; column < a[ 2 ].length; column++)
   a[ 2 ][ column ] = 0;
```

We specified row `2`; therefore, we know that the first index is always `2` (`0` is the first row, and `1` is the second row). This `for` loop varies only the second index (i.e., the column index). If row `2` of array `a` contains four elements, then the preceding `for` statement is equivalent to the assignment statements

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

The following nested `for` statement totals the values of all the elements in array `a`:

```
int total = 0;

for ( int row = 0; row < a.length; row++ )
{
   for ( int column = 0; column < a[ row ].length; column++ )
      total += a[ row ][ column ];
} // end outer for
```

These nested `for` statements total the array elements one row at a time. The outer `for` statement begins by setting the `row` index to `0` so that the first row's elements can be totaled by the inner `for` statement. The outer `for` then increments `row` to `1` so that the second row can be totaled. Then, the outer `for` increments `row` to `2` so that the third row can be totaled. The variable `total` can be displayed when the outer `for` statement terminates. In the next example, we show how to process a two-dimensional array in a similar manner using nested enhanced `for` statements.

E.10 Case Study: Class GradeBook Using a Two-Dimensional Array

In Section E.8, we presented class `GradeBook` (Fig. E.13), which used a one-dimensional array to store student grades on a single exam. In most semesters, students take several exams. Professors are likely to want to analyze grades across the entire semester, both for a single student and for the class as a whole.

Storing Student Grades in a Two-Dimensional Array in Class **GradeBook**

Figure E.17 contains a `GradeBook` class that uses a two-dimensional array `grades` to store the grades of a number of students on multiple exams. Each row of the array represents a single student's grades for the entire course, and each column represents the grades of all the students who took a particular exam. Class `GradeBookTest` (Fig. E.18) passes the array as an argument to the `GradeBook` constructor. In this example, we use a ten-by-three array for ten students' grades on three exams. Five methods perform array manipulations to process the grades. Each method is similar to its counterpart in the earlier one-dimensional array version of `GradeBook` (Fig. E.13). Method `getMinimum` (lines 52–70) determines the lowest grade of any student for the semester. Method `getMaximum` (lines 73–91) determines the highest grade of any student for the semester. Method `getAverage` (lines 94–104) determines a particular student's semester average. Method `outputBarChart` (lines 107–137) outputs a grade bar chart for the entire semester's student grades. Method `outputGrades` (lines 140–164) outputs the array in a tabular format, along with each student's semester average.

```
 1  // Fig. E.17: GradeBook.java
 2  // GradeBook class using a two-dimensional array to store grades.
 3
 4  public class GradeBook
 5  {
 6     private String courseName; // name of course this grade book represents
 7     private int[][] grades; // two-dimensional array of student grades
 8
 9     // two-argument constructor initializes courseName and grades array
10     public GradeBook( String name, int[][] gradesArray )
11     {
12        courseName = name; // initialize courseName
13        grades = gradesArray; // store grades
14     } // end two-argument GradeBook constructor
15
16     // method to set the course name
17     public void setCourseName( String name )
18     {
19        courseName = name; // store the course name
20     } // end method setCourseName
21
22     // method to retrieve the course name
23     public String getCourseName()
24     {
25        return courseName;
26     } // end method getCourseName
```

Fig. E.17 | `GradeBook` class using a two-dimensional array to store grades. (Part 1 of 4.)

```java

   // display a welcome message to the GradeBook user
   public void displayMessage()
   {
      // getCourseName gets the name of the course
      System.out.printf( "Welcome to the grade book for\n%s!\n\n",
         getCourseName() );
   } // end method displayMessage

   // perform various operations on the data
   public void processGrades()
   {
      // output grades array
      outputGrades();

      // call methods getMinimum and getMaximum
      System.out.printf( "\n%s %d\n%s %d\n\n",
         "Lowest grade in the grade book is", getMinimum(),
         "Highest grade in the grade book is", getMaximum() );

      // output grade distribution chart of all grades on all tests
      outputBarChart();
   } // end method processGrades

   // find minimum grade
   public int getMinimum()
   {
      // assume first element of grades array is smallest
      int lowGrade = grades[ 0 ][ 0 ];

      // loop through rows of grades array
      for ( int[] studentGrades : grades )
      {
         // loop through columns of current row
         for ( int grade : studentGrades )
         {
            // if grade less than lowGrade, assign it to lowGrade
            if ( grade < lowGrade )
               lowGrade = grade;
         } // end inner for
      } // end outer for

      return lowGrade; // return lowest grade
   } // end method getMinimum

   // find maximum grade
   public int getMaximum()
   {
      // assume first element of grades array is largest
      int highGrade = grades[ 0 ][ 0 ];

```

Fig. E.17 | GradeBook class using a two-dimensional array to store grades. (Part 2 of 4.)

```
      // loop through rows of grades array
      for ( int[] studentGrades : grades )
      {
         // loop through columns of current row
         for ( int grade : studentGrades )
         {
            // if grade greater than highGrade, assign it to highGrade
            if ( grade > highGrade )
               highGrade = grade;
         } // end inner for
      } // end outer for

      return highGrade; // return highest grade
   } // end method getMaximum

   // determine average grade for particular set of grades
   public double getAverage( int[] setOfGrades )
   {
      int total = 0; // initialize total

      // sum grades for one student
      for ( int grade : setOfGrades )
         total += grade;

      // return average of grades
      return (double) total / setOfGrades.length;
   } // end method getAverage

   // output bar chart displaying overall grade distribution
   public void outputBarChart()
   {
      System.out.println( "Overall grade distribution:" );

      // stores frequency of grades in each range of 10 grades
      int[] frequency = new int[ 11 ];

      // for each grade in GradeBook, increment the appropriate frequency
      for ( int[] studentGrades : grades )
      {
         for ( int grade : studentGrades )
            ++frequency[ grade / 10 ];
      } // end outer for

      // for each grade frequency, print bar in chart
      for ( int count = 0; count < frequency.length; count++ )
      {
         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
         if ( count == 10 )
            System.out.printf( "%5d: ", 100 );
         else
            System.out.printf( "%02d-%02d: ",
               count * 10, count * 10 + 9  );

```

Fig. E.17 | GradeBook class using a two-dimensional array to store grades. (Part 3 of 4.)

```
         // print bar of asterisks
         for ( int stars = 0; stars < frequency[ count ]; stars++ )
            System.out.print( "*" );

         System.out.println(); // start a new line of output
      } // end outer for
   } // end method outputBarChart

   // output the contents of the grades array
   public void outputGrades()
   {
      System.out.println( "The grades are:\n" );
      System.out.print( "            " ); // align column heads

      // create a column heading for each of the tests
      for ( int test = 0; test < grades[ 0 ].length; test++ )
         System.out.printf( "Test %d  ", test + 1 );

      System.out.println( "Average" ); // student average column heading

      // create rows/columns of text representing array grades
      for ( int student = 0; student < grades.length; student++ )
      {
         System.out.printf( "Student %2d", student + 1 );

         for ( int test : grades[ student ] ) // output student's grades
            System.out.printf( "%8d", test );

         // call method getAverage to calculate student's average grade;
         // pass row of grades as the argument to getAverage
         double average = getAverage( grades[ student ] );
         System.out.printf( "%9.2f\n", average );
      } // end outer for
   } // end method outputGrades
} // end class GradeBook
```

Fig. E.17 | GradeBook class using a two-dimensional array to store grades. (Part 4 of 4.)

Methods **getMinimum** *and* **getMaximum**

Methods getMinimum, getMaximum, outputBarChart and outputGrades each loop through array grades by using nested for statements—for example, the nested enhanced for statement from the declaration of method getMinimum (lines 58–67). The outer enhanced for statement iterates through the two-dimensional array grades, assigning successive rows to parameter studentGrades on successive iterations. The square brackets following the parameter name indicate that studentGrades refers to a one-dimensional int array—namely, a row in array grades containing one student's grades. To find the lowest overall grade, the inner for statement compares the elements of the current one-dimensional array studentGrades to variable lowGrade. For example, on the first iteration of the outer for, row 0 of grades is assigned to parameter studentGrades. The inner enhanced for statement then loops through studentGrades and compares each grade value with lowGrade. If a grade is less than lowGrade, lowGrade is set to that grade. On the second iteration of the outer enhanced `for` statement, row 1 of `grades` is assigned to `studentGrades`, and the elements of this row are compared with variable `lowGrade`. This repeats until all rows of `grades` have been traversed. When execution of the nested statement is complete, `lowGrade` contains the lowest grade in the two-dimensional array. Method `getMaximum` works similarly to method `getMinimum`.

Method `outputBarChart`

Method `outputBarChart` (lines 107–137) is nearly identical to the one in Fig. E.13. However, to output the overall grade distribution for a whole semester, the method here uses nested enhanced `for` statements (lines 115–119) to create the one-dimensional array `frequency` based on all the grades in the two-dimensional array. The rest of the code in each of the two `outputBarChart` methods that displays the chart is identical.

Method `outputGrades`

Method `outputGrades` (lines 140–164) uses nested `for` statements to output values of the array `grades` and each student's semester average. The output (Fig. E.18) shows the result, which resembles the tabular format of a professor's physical grade book. Lines 146–147 print the column headings for each test. We use a counter-controlled `for` statement here so that we can identify each test with a number. Similarly, the `for` statement in lines 152–163 first outputs a row label using a counter variable to identify each student (line 154). Although array indices start at 0, lines 147 and 154 output `test + 1` and `student + 1`, respectively, to produce test and student numbers starting at 1 (see Fig. E.18). The inner `for` statement (lines 156–157) uses the outer `for` statement's counter variable `student` to loop through a specific row of array `grades` and output each student's test grade. An enhanced `for` statement can be nested in a counter-controlled `for` statement, and vice versa. Finally, line 161 obtains each student's semester average by passing the current row of `grades` (i.e., `grades[student]`) to method `getAverage`.

Method `getAverage`

Method `getAverage` (lines 94–104) takes one argument—a one-dimensional array of test results for a particular student. When line 161 calls `getAverage`, the argument is `grades[student]`, which specifies that a particular row of the two-dimensional array `grades` should be passed to `getAverage`. For example, based on the array created in Fig. E.18, the argument `grades[1]` represents the three values (a one-dimensional array of grades) stored in row `1` of the two-dimensional array `grades`. Recall that a two-dimensional array is one whose elements are one-dimensional arrays. Method `getAverage` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result as a `double` value (line 103).

Class `GradeBookTest` *That Demonstrates Class* `GradeBook`

Figure E.18 creates an object of class `GradeBook` (Fig. E.17) using the two-dimensional array of `int`s named `gradesArray` (declared and initialized in lines 10–19). Lines 21–22 pass a course name and `gradesArray` to the `GradeBook` constructor. Lines 23–24 then invoke `myGradeBook`'s `displayMessage` and `processGrades` methods to display a welcome message and obtain a report summarizing the students' grades for the semester, respectively.

```java
// Fig. E.18: GradeBookTest.java
// GradeBookTest creates GradeBook object using a two-dimensional array
// of grades, then invokes method processGrades to analyze them.
public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // two-dimensional array of student grades
      int[][] gradesArray = { { 87, 96, 70 },
                              { 68, 87, 90 },
                              { 94, 100, 90 },
                              { 100, 81, 82 },
                              { 83, 65, 85 },
                              { 78, 87, 65 },
                              { 85, 75, 83 },
                              { 91, 94, 100 },
                              { 76, 72, 84 },
                              { 87, 93, 73 } };

      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to Java Programming", gradesArray );
      myGradeBook.displayMessage();
      myGradeBook.processGrades();
   } // end main
} // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
```

Fig. E.18 | GradeBookTest creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 1 of 2.)

```
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. E.18 | `GradeBookTest` creates `GradeBook` object using a two-dimensional array of grades, then invokes method `processGrades` to analyze them. (Part 2 of 2.)

E.11 Class Arrays

Class **`Arrays`** helps you avoid reinventing the wheel by providing `static` methods for common array manipulations. These methods include **`sort`** for sorting an array (i.e., arranging elements into increasing order), **`binarySearch`** for searching an array (i.e., determining whether an array contains a specific value and, if so, where the value is located), **`equals`** for comparing arrays and **`fill`** for placing values into an array. These methods are overloaded for primitive-type arrays and for arrays of objects. Our focus in this section is on using the built-in capabilities provided by the Java API.

Figure E.19 uses `Arrays` methods `sort`, `binarySearch`, `equals` and `fill`, and shows how to copy arrays with class `System`'s `static` **`arraycopy` method**. In `main`, line 11 sorts the elements of array `doubleArray`. The `static` method `sort` of class `Arrays` orders the array's elements in *ascending* order by default. Overloaded versions of `sort` allow you to sort a specific range of elements. Lines 12–15 output the sorted array.

```
 1  // Fig. E.19: ArrayManipulations.java
 2  // Arrays class methods and System.arraycopy.
 3  import java.util.Arrays;
 4
 5  public class ArrayManipulations
 6  {
 7     public static void main( String[] args )
 8     {
 9        // sort doubleArray into ascending order
10        double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11        Arrays.sort( doubleArray );
12        System.out.printf( "\ndoubleArray: " );
13
14        for ( double value : doubleArray )
15           System.out.printf( "%.1f ", value );
16
17        // fill 10-element array with 7s
18        int[] filledIntArray = new int[ 10 ];
19        Arrays.fill( filledIntArray, 7 );
20        displayArray( filledIntArray, "filledIntArray" );
21
```

Fig. E.19 | `Arrays` class methods and `System.arraycopy`. (Part 1 of 3.)

```
      // copy array intArray into array intArrayCopy
      int[] intArray = { 1, 2, 3, 4, 5, 6 };
      int[] intArrayCopy = new int[ intArray.length ];
      System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length );
      displayArray( intArray, "intArray" );
      displayArray( intArrayCopy, "intArrayCopy" );

      // compare intArray and intArrayCopy for equality
      boolean b = Arrays.equals( intArray, intArrayCopy );
      System.out.printf( "\n\nintArray %s intArrayCopy\n",
         ( b ? "==" : "!=" ) );

      // compare intArray and filledIntArray for equality
      b = Arrays.equals( intArray, filledIntArray );
      System.out.printf( "intArray %s filledIntArray\n",
         ( b ? "==" : "!=" ) );

      // search intArray for the value 5
      int location = Arrays.binarySearch( intArray, 5 );

      if ( location >= 0 )
         System.out.printf(
            "Found 5 at element %d in intArray\n", location );
      else
         System.out.println( "5 not found in intArray" );

      // search intArray for the value 8763
      location = Arrays.binarySearch( intArray, 8763 );

      if ( location >= 0 )
         System.out.printf(
            "Found 8763 at element %d in intArray\n", location );
      else
         System.out.println( "8763 not found in intArray" );
   } // end main

   // output values in each array
   public static void displayArray( int[] array, String description )
   {
      System.out.printf( "\n%s: ", description );

      for ( int value : array )
         System.out.printf( "%d ", value );
   } // end method displayArray
} // end class ArrayManipulations
```

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6
```

Fig. E.19 | Arrays class methods and System.arraycopy. (Part 2 of 3.)

```
intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

Fig. E.19 | `Arrays` class methods and `System.arraycopy`. (Part 3 of 3.)

Line 19 calls `static` method `fill` of class `Arrays` to populate all 10 elements of `filledIntArray` with 7s. Overloaded versions of `fill` allow you to populate a specific range of elements with the same value. Line 20 calls our class's `displayArray` method (declared at lines 59–65) to output the contents of `filledIntArray`.

Line 25 copies the elements of `intArray` into `intArrayCopy`. The first argument (`intArray`) passed to `System` method `arraycopy` is the array from which elements are to be copied. The second argument (`0`) is the index that specifies the starting point in the range of elements to copy from the array. This value can be any valid array index. The third argument (`intArrayCopy`) specifies the destination array that will store the copy. The fourth argument (`0`) specifies the index in the destination array where the first copied element should be stored. The last argument specifies the number of elements to copy from the array in the first argument. In this case, we copy all the elements in the array.

Lines 30 and 35 call `static` method `equals` of class `Arrays` to determine whether all the elements of two arrays are equivalent. If the arrays contain the same elements in the same order, the method returns `true`; otherwise, it returns `false`.

Lines 40 and 49 call `static` method `binarySearch` of class `Arrays` to perform a binary search on `intArray`, using the second argument (`5` and `8763`, respectively) as the key. If `value` is found, `binarySearch` returns the index of the element; otherwise, `binarySearch` returns a negative value. The negative value returned is based on the search key's insertion point—the index where the key would be inserted in the array if we were performing an insert operation. After `binarySearch` determines the insertion point, it changes its sign to negative and subtracts `1` to obtain the return value. For example, in Fig. E.19, the insertion point for the value `8763` is the element with index `6` in the array. Method `binarySearch` changes the insertion point to `-6`, subtracts `1` from it and returns the value `-7`. Subtracting `1` from the insertion point guarantees that method `binarySearch` returns positive values (>= 0) if and only if the key is found. This return value is useful for inserting elements in a sorted array.

Common Programming Error E.3

Passing an unsorted array to `binarySearch` is a logic error—the value returned is undefined.

E.12 Introduction to Collections and Class `ArrayList`

The Java API provides several predefined data structures, called **collections**, used to store groups of related objects. These classes provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored. This reduces application-development time.

You've used arrays to store sequences of objects. Arrays do not automatically change their size at execution time to accommodate additional elements. The collection class

`ArrayList<T>` (from package `java.util`) provides a convenient solution to this problem—it can *dynamically* change its size to accommodate more elements. The `T` (by convention) is a *placeholder*—when declaring a new `ArrayList`, replace it with the type of elements that you want the `ArrayList` to hold. This is similar to specifying the type when declaring an array, except that *only nonprimitive types can be used with these collection classes*. For example,

```
    ArrayList< String > list;
```

declares `list` as an `ArrayList` collection that can store only `String`s. Classes with this kind of placeholder that can be used with any type are called **generic classes**. Additional generic collection classes and generics are discussed in Appendix J. Figure E.20 shows some common methods of class `ArrayList<T>`.

Method	Description
`add`	Adds an element to the end of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements.

Fig. E.20 | Some methods and properties of class `ArrayList<T>`.

Figure E.21 demonstrates some common `ArrayList` capabilities. Line 10 creates a new empty `ArrayList` of `String`s with a default initial capacity of 10 elements. The capacity indicates how many items the `ArrayList` can hold without growing. `ArrayList` is implemented using an array behind the scenes. When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array. This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added. Instead, it grows only when an element is added *and* the number of elements is equal to the capacity—i.e., there's no space for the new element.

```
1  // Fig. E.21: ArrayListCollection.java
2  // Generic ArrayList<T> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
```

Fig. E.21 | Generic `ArrayList<T>` collection demonstration. (Part 1 of 3.)

```
    public static void main( String[] args )
    {
       // create a new ArrayList of Strings with an initial capacity of 10
       ArrayList< String > items = new ArrayList< String >();

       items.add( "red" ); // append an item to the list
       items.add( 0, "yellow" ); // insert the value at index 0

       // header
       System.out.print(
          "Display list contents with counter-controlled loop:" );

       // display the colors in the list
       for ( int i = 0; i < items.size(); i++ )
          System.out.printf( " %s", items.get( i ) );

       // display colors using foreach in the display method
       display( items,
          "\nDisplay list contents with enhanced for statement:" );

       items.add( "green" ); // add "green" to the end of the list
       items.add( "yellow" ); // add "yellow" to the end of the list
       display( items, "List with two new elements:" );

       items.remove( "yellow" ); // remove the first "yellow"
       display( items, "Remove first instance of yellow:" );

       items.remove( 1 ); // remove item at index 1
       display( items, "Remove second list element (green):" );

       // check if a value is in the List
       System.out.printf( "\"red\" is %sin the list\n",
          items.contains( "red" ) ? "": "not " );

       // display number of elements in the List
       System.out.printf( "Size: %s\n", items.size() );
    } // end main

    // display the ArrayList's elements on the console
    public static void display( ArrayList< String > items, String header )
    {
       System.out.print( header ); // display header

       // display each element in items
       for ( String item : items )
          System.out.printf( " %s", item );

       System.out.println(); // display end of line
    } // end method display
 } // end class ArrayListCollection
```

Fig. E.21 | Generic `ArrayList<T>` collection demonstration. (Part 2 of 3.)

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Fig. E.21 | Generic `ArrayList<T>` collection demonstration. (Part 3 of 3.)

The **`add`** method adds elements to the `ArrayList` (lines 12–13). The `add` method with *one* argument appends its argument to the end of the `ArrayList`. The `add` method with *two* arguments inserts a new element at the specified position. The first argument is an index. As with arrays, collection indices start at zero. The second argument is the value to insert at that index. The indices of all subsequent elements are incremented by one. Inserting an element is usually slower than adding an element to the end of the `ArrayList`

Lines 20–21 display the items in the `ArrayList`. The **`size`** method returns the number of elements currently in the `ArrayList`. `ArrayList`s method **`get`** (line 21) obtains the element at a specified index. Lines 24–25 display the elements again by invoking method display (defined at lines 46–55). Lines 27–28 add two more elements to the `ArrayList`, then line 29 displays the elements again to confirm that the two elements were added to the end of the collection.

The **`remove`** method is used to remove an element with a specific value (line 31). It removes only the first such element. If no such element is in the `ArrayList`, `remove` does nothing. An overloaded version of the method removes the element at the specified index (line 34). When an element is removed, the indices of all elements after the removed element decrease by one.

Line 39 uses the **`contains`** method to check if an item is in the `ArrayList`. The `contains` method returns `true` if the element is found in the `ArrayList`, and `false` otherwise. The method compares its argument to each element of the `ArrayList` in order, so using `contains` on a large `ArrayList` can be inefficient. Line 42 displays the `ArrayList`'s size.

E.13 Wrap-Up

This appendix began our introduction to data structures, exploring the use of arrays to store data in and retrieve data from lists and tables of values. The appendix examples demonstrated how to declare an array, initialize an array and refer to individual elements of an array. The appendix introduced the enhanced `for` statement to iterate through arrays. We used exception handling to test for `ArrayIndexOutOfBoundsException`s that occur when a program attempts to access an array element outside the bounds of an array. We also illustrated how to pass arrays to methods and how to declare and manipulate multidimensional arrays.

We introduced the `ArrayList<T>` generic collection, which provides all the functionality and performance of arrays, along with other useful capabilities such as dynamic resizing. We used the `add` methods to add new items to the end of an `ArrayList` and to insert items in an `ArrayList`. The `remove` method was used to remove the first occurrence of a specified item, and an overloaded version of `remove` was used to remove an item at a specified index. We used the `size` method to obtain number of items in the `ArrayList`.

We continue our coverage of data structures in Appendix J. Appendix J introduces the Java Collections Framework, which uses generics to allow you to specify the exact types of objects that a particular data structure will store. Appendix J also introduces Java's other predefined data structures. The Collections API provides class `Arrays`, which contains utility methods for array manipulation. Appendix J uses several `static` methods of class `Arrays` to perform such manipulations as sorting and searching the data in an array.

We've now introduced the basic concepts of classes, objects, control statements, methods, arrays and collections. In Appendix F, we take a deeper look at classes and objects.

Self-Review Exercises

E.1 Fill in the blank(s) in each of the following statements:

a) Lists and tables of values can be stored in ________.
b) Any attempt to access an element outside a particular range of array element indices results in a runtime error that's known as an ________.
c) The ________ allows you to iterate through the elements in an array without using a counter.
d) The number used to refer to a particular array element is called the element's ________.
e) An array that uses two indices is referred to as a(n) ________ array.
f) Use the enhanced `for` statement ________ to walk through `double` array `numbers`.
g) Command-line arguments are stored in ________.

E.2 Determine whether each of the following is *true* or *false*. If *false*, explain why.

a) An array can store many different types of values.
b) An array index should normally be of type `float`.
c) An individual array element that's passed to a method and modified in that method will contain the modified value when the called method completes execution.

E.3 Perform the following tasks for an array called `numbers`:

a) Declare a constant `ARRAY_SIZE` that's initialized to 12.
b) Declare an array with `ARRAY_SIZE` elements of type `int`, and initialize the elements to 0.
c) Refer to array element 6.
d) Assign the value `6` to array element 9.
e) Assign the value `8` to array element 6.
f) Sum all the elements of the array, using a `for` statement. Declare the integer variable `i` as a control variable for the loop.

E.4 Perform the following tasks for an array called `table`:

a) Declare and create the array as an integer array that has three rows and three columns. Assume that the constant `ARRAY_SIZE` has been declared to be `3`.
b) How many elements does the array contain?
c) Use a `for` statement to initialize each element of the array to the sum of its indices. Assume that the integer variables `x` and `y` are declared as control variables.

E.5 Find and correct the error in each of the following program segments:

a)
```
final int ARRAY_SIZE = 5;
ARRAY_SIZE = 10;
```
b) Assume `int[] b = new int[ 10 ];`
```
for ( int i = 0; i <= b.length; i++ )
   b[ i ] = 1;
```
c) Assume `int[][] a = { { 1, 2 }, { 3, 4 } };`
```
   a[ 1, 1 ] = 5;
```

Answers to Self-Review Exercises

E.1 a) arrays. b) `ArrayIndexOutOfBoundsException`. c) enhanced `for` statement. d) index (or subscript or position number). e) two-dimensional. f) `for ( double d : numbers )`. g) an array of `String`s, called `args` by convention.

E.2 a) False. An array can store only values of the same type. b) False. An array index must be an integer or an integer expression. c) For individual primitive-type elements of an array: False. A called method receives and manipulates a copy of the value of such an element, so modifications do not affect the original value. If the reference of an array is passed to a method, however, modifications to the array elements made in the called method are indeed reflected in the original. For individual elements of a reference type: True. A called method receives a copy of the reference of such an element, and changes to the referenced object will be reflected in the original array element.

E.3
a) `final int ARRAY_SIZE = 12;`
b) `int[] numbers = new int[ ARRAY_SIZE ];`
c) `numbers[ 6 ]`
d) `numbers[ 9 ] = 6;`
e) `numbers[ 6 ] = 8;`
f)
```
int total = 0;
for ( int i = 0; i < numbers.length; i++ )
   total += numbers[ i ];
```

E.4
a) `int[][] table = new int[ ARRAY_SIZE ][ ARRAY_SIZE ];`
b) Nine.
c)
```
for ( int x = 0; x < table.length; x++ )
   for ( int y = 0; y < table[ x ].length; y++ )
      table[ x ][ y ] = x + y;
```

E.5
a) Error: Assigning a value to a constant after it has been initialized.
Correction: Assign the correct value to the constant in a `final int ARRAY_SIZE` declaration or declare another variable.
b) Error: Referencing an array element outside the bounds of the array (`b[10]`).
Correction: Change the `<=` operator to `<`.
c) Error: Array indexing is performed incorrectly.
Correction: Change the statement to `a[ 1 ][ 1 ] = 5;`.

Exercises

E.6 Fill in the blanks in each of the following statements:
a) In the loop-continuation condition of the outer `for` statement, the expression ________ determines the number of rows in the array.
b) By providing `static` methods for common array manipulations, class ________ helps you avoid reinventing the wheel.

c) In a two-dimensional array, the first index identifies the ________ of an element and the second index identifies the ________ of an element.
d) An *m*-by-*n* array contains ________ rows, ________ columns and ________ elements.
e) The name of the element in row 3 and column 5 of array `d` is ________.

E.7 Determine whether each of the following is *true* or *false*. If *false*, explain why.

a) To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.
b) An array declaration reserves space for the array.
c) To indicate that 100 locations should be reserved for integer array `p`, you write the declaration

```
p[ 100 ];
```

d) An application that initializes the elements of a 15-element array to zero must contain at least one `for` statement.
e) An application that totals the elements of a two-dimensional array must contain nested `for` statements.

E.8 Consider a two-by-three integer array `t`.

a) Write a statement that declares and creates `t`.
b) How many rows does `t` have?
c) How many columns does `t` have?
d) How many elements does `t` have?
e) Write access expressions for all the elements in row 1 of `t`.
f) Write access expressions for all the elements in column 2 of `t`.
g) Write a single statement that sets the element of `t` in row 0 and column 1 to zero.
h) Write individual statements to initialize each element of `t` to zero.
i) Write a nested `for` statement that initializes each element of `t` to zero.
j) Write a nested `for` statement that inputs the values for the elements of `t` from the user.
k) Write a series of statements that determines and displays the smallest value in `t`.
l) Write a single `printf` statement that displays the elements of the first row of `t`.
m) Write a statement that totals the elements of the third column of `t`. Do not use repetition.
n) Write a series of statements that displays the contents of `t` in tabular format. List the column indices as headings across the top, and list the row indices at the left of each row.

E.9 *(Duplicate Elimination)* Use a one-dimensional array to solve the following problem: Write an application that inputs five numbers, each between 10 and 100, inclusive. As each number is read, display it only if it's not a duplicate of a number already read. Provide for the "worst case," in which all five numbers are different. Use the smallest possible array to solve this problem. Display the complete set of unique values input after the user enters each new value.

E.10 Label the elements of three-by-five two-dimensional array `sales` to indicate the order in which they're set to zero by the following program segment:

```
for ( int row = 0; row < sales.length; row++ )
{
   for ( int col = 0; col < sales[ row ].length; col++ )
   {
      sales[ row ][ col ] = 0;
   }
}
```

E.11 *(Sieve of Eratosthenes)* A prime number is any integer greater than 1 that's evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a) Create a primitive-type `boolean` array with all elements initialized to `true`. Array elements with prime indices will remain `true`. All other array elements will eventually be set to `false`.
b) Starting with array index 2, determine whether a given element is `true`. If so, loop through the remainder of the array and set to `false` every element whose index is a multiple of the index for the element with value `true`. Then continue the process with the next element with value `true`. For array index 2, all elements beyond element 2 in the array that have indices which are multiples of 2 (indices 4, 6, 8, 10, etc.) will be set to `false`; for array index 3, all elements beyond element 3 in the array that have indices which are multiples of 3 (indices 6, 9, 12, 15, etc.) will be set to `false`; and so on.

When this process completes, the array elements that are still `true` indicate that the index is a prime number. These indices can be displayed. Write an application that uses an array of 1000 elements to determine and display the prime numbers between 2 and 999. Ignore array elements 0 and 1.

E.12 *(Fibonacci Series)* The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms.

a) Write a method `fibonacci( n )` that calculates the *n*th Fibonacci number. Incorporate this method into an application that enables the user to enter the value of `n`.
b) Determine the largest Fibonacci number that can be displayed on your system.
c) Modify the application you wrote in part (a) to use `double` instead of `int` to calculate and return Fibonacci numbers, and use this modified application to repeat part (b).

F

Classes and Objects: A Deeper Look

Objectives

In this appendix you'll learn:

- Encapsulation and data hiding.
- To use keyword `this`.
- To use `static` variables and methods.
- To import `static` members of a class.
- To use the `enum` type to create sets of constants with unique identifiers.
- To declare `enum` constants with parameters.
- To organize classes in packages to promote reuse.

Outline

F.1 Introduction

We now take a deeper look at building classes, controlling access to members of a class and creating constructors. We discuss composition—a capability that allows a class to have references to objects of other classes as members. Recall that Section D.10 introduced the basic `enum` type to declare a set of constants. In this appendix, we discuss the relationship between `enum` types and classes, demonstrating that an `enum`, like a class, can be declared in its own file with constructors, methods and fields. The appendix also discusses `static` class members and `final` instance variables in detail. Finally, we explain how to organize classes in packages to help manage large applications and promote reuse, then show a special relationship between classes in the same package.

F.2 `Time` Class Case Study

Our first example consists of two classes—`Time1` (Fig. F.1) and `Time1Test` (Fig. F.2). Class `Time1` represents the time of day. Class `Time1Test` is an application class in which the `main` method creates one object of class `Time1` and invokes its methods. These classes must be declared in *separate* files because they're both `public` classes. The output of this program appears in Fig. F.2.

`Time1` Class Declaration

Class `Time1`'s `private int` instance variables `hour`, `minute` and `second` (Fig. F.1, lines 6–8) represent the time in universal-time format (24-hour clock format in which hours are in the range 0–23). Class `Time1` contains `public` methods `setTime` (lines 12–25), `toUniversalString` (lines 28–31) and `toString` (lines 34–39). These methods are also called the **`public` services** or the **`public` interface** that the class provides to its clients.

Default Constructor

In this example, class `Time1` does not declare a constructor, so the class has a default constructor that's supplied by the compiler. Each instance variable implicitly receives the default value `0` for an `int`. Instance variables also can be initialized when they're declared in the class body, using the same initialization syntax as with a local variable.

```
 1  // Fig. F.1: Time1.java
 2  // Time1 class declaration maintains the time in 24-hour format.
 3
 4  public class Time1
 5  {
 6     private int hour; // 0 - 23
 7     private int minute; // 0 - 59
 8     private int second; // 0 - 59
 9
10     // set a new time value using universal time; throw an
11     // exception if the hour, minute or second is invalid
12     public void setTime( int h, int m, int s )
13     {
14        // validate hour, minute and second
15        if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
16           ( s >= 0 && s < 60 ) )
17        {
18           hour = h;
19           minute = m;
20           second = s;
21        } // end if
22        else
23           throw new IllegalArgumentException(
24              "hour, minute and/or second was out of range" );
25     } // end method setTime
26
27     // convert to String in universal-time format (HH:MM:SS)
28     public String toUniversalString()
29     {
30        return String.format( "%02d:%02d:%02d", hour, minute, second );
31     } // end method toUniversalString
32
33     // convert to String in standard-time format (H:MM:SS AM or PM)
34     public String toString()
35     {
36        return String.format( "%d:%02d:%02d %s",
37           ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
38           minute, second, ( hour < 12 ? "AM" : "PM" ) );
39     } // end method toString
40  } // end class Time1
```

Fig. F.1 | `Time1` class declaration maintains the time in 24-hour format.

Method `setTime` and Throwing Exceptions

Method `setTime` (lines 12–25) is a `public` method that declares three `int` parameters and uses them to set the time. Lines 15–16 test each argument to determine whether the value is in the proper range, and, if so, lines 18–20 assign the values to the `hour`, `minute` and `second` instance variables. The `hour` value must be greater than or equal to `0` and less than `24`, because universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13 and 11 PM is hour 23; midnight is hour 0 and noon is hour 12). Similarly, both `minute` and `second` values must be greater than or equal to `0` and less than `60`. For values outside these ranges, `SetTime` **throws an exception** of type **`IllegalArgumentException`** (lines 23–24), which notifies the client code that an invalid argument was passed to the method. As you learned in Appendix E, you can use try...catch to catch exceptions and attempt to recover from them, which we'll do in Fig. F.2. The **throw statement** (line 23) creates a new object of type IllegalArgumentException. The parentheses following the class name indicate a call to the IllegalArgumentException constructor. In this case, we call the constructor that allows us to specify a custom error message. After the exception object is created, the throw statement immediately terminates method setTime and the exception is returned to the code that attempted to set the time.

Method **toUniversalString**

Method toUniversalString (lines 28–31) takes no arguments and returns a String in universal-time format, consisting of two digits each for the hour, minute and second. For example, if the time were 1:30:07 PM, the method would return 13:30:07. Line 30 uses static method **format** of class String to return a String containing the formatted hour, minute and second values, each with two digits and possibly a leading 0 (specified with the 0 flag). Method format is similar to method System.out.printf except that format *returns* a formatted String rather than displaying it in a command window. The formatted String is returned by method toUniversalString.

Method **toString**

Method toString (lines 34–39) takes no arguments and returns a String in standard-time format, consisting of the hour, minute and second values separated by colons and followed by AM or PM (e.g., 1:27:06 PM). Like method toUniversalString, method toString uses static String method format to format the minute and second as two-digit values, with leading zeros if necessary. Line 37 uses a conditional operator (?:) to determine the value for hour in the String—if the hour is 0 or 12 (AM or PM), it appears as 12; otherwise, it appears as a value from 1 to 11. The conditional operator in line 38 determines whether AM or PM will be returned as part of the String.

Recall from Section D.4 that all objects in Java have a toString method that returns a String representation of the object. We chose to return a String containing the time in standard-time format. Method toString is called implicitly whenever a Time1 object appears in the code where a String is needed, such as the value to output with a %s format specifier in a call to System.out.printf.

Using Class **Time1**

As you learned in Appendix B, each class you declare represents a new *type* in Java. Therefore, after declaring class Time1, we can use it as a type in declarations such as

```
    Time1 sunset; // sunset can hold a reference to a Time1 object
```

The Time1Test application class (Fig. F.2) uses class Time1. Line 9 declares and creates a Time1 object and assigns it to local variable time. Operator new implicitly invokes class Time1's default constructor, since Time1 does not declare any constructors. Lines 12–16 output the time first in universal-time format (by invoking time's toUniversalString method in line 13), then in standard-time format (by explicitly invoking time's toString method in line 15) to confirm that the Time1 object was initialized properly. Next, line 19 invokes method setTime of the time object to change the time. Then lines 20–24 output the time again in both formats to confirm that it was set correctly.

```java
// Fig. F.2: Time1Test.java
// Time1 object used in an application.

public class Time1Test
{
   public static void main( String[] args )
   {
      // create and initialize a Time1 object
      Time1 time = new Time1(); // invokes Time1 constructor

      // output string representations of the time
      System.out.print( "The initial universal time is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "The initial standard time is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

      // change time and output updated time
      time.setTime( 13, 27, 6 );
      System.out.print( "Universal time after setTime is: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time after setTime is: " );
      System.out.println( time.toString() );
      System.out.println(); // output a blank line

      // attempt to set time with invalid values
      try
      {
         time.setTime( 99, 99, 99 ); // all values out of range
      } // end try
      catch ( IllegalArgumentException e )
      {
         System.out.printf( "Exception: %s\n\n", e.getMessage() );
      } // end catch

      // display time after attempt to set invalid values
      System.out.println( "After attempting invalid settings:" );
      System.out.print( "Universal time: " );
      System.out.println( time.toUniversalString() );
      System.out.print( "Standard time: " );
      System.out.println( time.toString() );
   } // end main
} // end class Time1Test
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting invalid settings:
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Fig. F.2 | Time1 object used in an application.

Calling `Time1` Method `setTime` with Invalid Values

To illustrate that method `setTime` validates its arguments, line 29 calls method `setTime` with invalid arguments of `99` for the `hour`, `minute` and `second`. This statement is placed in a `try` block (lines 27–30) in case `setTime` throws an `IllegalArgumentException`, which it will do since the arguments are all invalid. When this occurs, the exception is caught at lines 31–34, and line 33 displays the exception's error message by calling its `getMessage` method. Lines 37–41 output the time again in both formats to confirm that `setTime` did not change the time when invalid arguments were supplied.

Notes on the `Time1` Class Declaration

Consider several issues of class design with respect to class `Time1`. The instance variables `hour`, `minute` and `second` are each declared `private`. The actual data representation used within the class is of no concern to the class's clients. For example, it would be perfectly reasonable for `Time1` to represent the time internally as the number of seconds since midnight or the number of minutes and seconds since midnight. Clients could use the same `public` methods and get the same results without being aware of this.

F.3 Controlling Access to Members

The access modifiers `public` and `private` control access to a class's variables and methods. In Appendix G, we'll introduce the access modifier `protected`. As you know, the primary purpose of `public` methods is to present to the class's clients a view of the services the class provides (the class's `public` interface). Clients need not be concerned with how the class accomplishes its tasks. For this reason, the class's `private` variables and `private` methods (i.e., its implementation details) are *not* accessible to its clients.

Figure F.3 demonstrates that `private` class members are not accessible outside the class. Lines 9–11 attempt to access directly the `private` instance variables `hour`, `minute` and `second` of the `Time1` object `time`. When this program is compiled, the compiler generates error messages that these `private` members are not accessible. This program assumes that the `Time1` class from Fig. F.1 is used.

```
 1  // Fig. F.3: MemberAccessTest.java
 2  // Private members of class Time1 are not accessible.
 3  public class MemberAccessTest
 4  {
 5     public static void main( String[] args )
 6     {
 7        Time1 time = new Time1(); // create and initialize Time1 object
 8
 9        time.hour = 7; // error: hour has private access in Time1
10        time.minute = 15; // error: minute has private access in Time1
11        time.second = 30; // error: second has private access in Time1
12     } // end main
13  } // end class MemberAccessTest
```

Fig. F.3 | Private members of class `Time1` are not accessible. (Part 1 of 2.)

```
MemberAccessTest.java:9: hour has private access in Time1
      time.hour = 7; // error: hour has private access in Time1
          ^
MemberAccessTest.java:10: minute has private access in Time1
      time.minute = 15; // error: minute has private access in Time1
          ^
MemberAccessTest.java:11: second has private access in Time1
      time.second = 30; // error: second has private access in Time1
          ^
3 errors
```

Fig. F.3 | Private members of class `Time1` are not accessible. (Part 2 of 2.)

F.4 Referring to the Current Object's Members with the `this` Reference

Every object can access a reference to itself with keyword **`this`** (sometimes called the **`this` reference**). When a non-`static` method is called for a particular object, the method's body implicitly uses keyword `this` to refer to the object's instance variables and other methods. This enables the class's code to know which object should be manipulated. As you'll see in Fig. F.4, you can also use keyword `this` explicitly in a non-`static` method's body. Section F.5 shows another interesting use of keyword `this`. Section F.10 explains why keyword `this` cannot be used in a `static` method.

We now demonstrate implicit and explicit use of the `this` reference (Fig. F.4). This example is the first in which we declare *two* classes in one file—class `ThisTest` is declared in lines 4–11, and class `SimpleTime` in lines 14–47. We do this to demonstrate that when you compile a `.java` file containing more than one class, the compiler produces a separate class file with the `.class` extension for every compiled class. In this case, two separate files are produced—`SimpleTime.class` and `ThisTest.class`. When one source-code (`.java`) file contains multiple class declarations, the compiler places both class files for those classes in the same directory. Note also in Fig. F.4 that only class `ThisTest` is declared `public`. A source-code file can contain only one `public` class—otherwise, a compilation error occurs. Non-`public` classes can be used only by other classes in the same package. So, in this example, class `SimpleTime` can be used only by class `ThisTest`.

```
 1  // Fig. F.4: ThisTest.java
 2  // this used implicitly and explicitly to refer to members of an object.
 3
 4  public class ThisTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        SimpleTime time = new SimpleTime( 15, 30, 19 );
 9        System.out.println( time.buildString() );
10     } // end main
11  } // end class ThisTest
12
```

Fig. F.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 1 of 2.)

```
13  // class SimpleTime demonstrates the "this" reference
14  class SimpleTime
15  {
16     private int hour; // 0-23
17     private int minute; // 0-59
18     private int second; // 0-59
19
20     // if the constructor uses parameter names identical to
21     // instance variable names the "this" reference is
22     // required to distinguish between the names
23     public SimpleTime( int hour, int minute, int second )
24     {
25        this.hour = hour; // set "this" object's hour
26        this.minute = minute; // set "this" object's minute
27        this.second = second; // set "this" object's second
28     } // end SimpleTime constructor
29
30     // use explicit and implicit "this" to call toUniversalString
31     public String buildString()
32     {
33        return String.format( "%24s: %s\n%24s: %s",
34           "this.toUniversalString()", this.toUniversalString(),
35           "toUniversalString()", toUniversalString() );
36     } // end method buildString
37
38     // convert to String in universal-time format (HH:MM:SS)
39     public String toUniversalString()
40     {
41        // "this" is not required here to access instance variables,
42        // because method does not have local variables with same
43        // names as instance variables
44        return String.format( "%02d:%02d:%02d",
45           this.hour, this.minute, this.second );
46     } // end method toUniversalString
47  } // end class SimpleTime
```

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

Fig. F.4 | this used implicitly and explicitly to refer to members of an object. (Part 2 of 2.)

Class SimpleTime (lines 14–47) declares three private instance variables—hour, minute and second (lines 16–18). The constructor (lines 23–28) receives three int arguments to initialize a SimpleTime object. We used parameter names for the constructor (line 23) that are identical to the class's instance-variable names (lines 16–18). We don't recommend this practice, but we did it here to shadow (hide) the corresponding instance variables so that we could illustrate a case in which *explicit* use of the this reference is required. If a method contains a local variable with the *same* name as a field, that method will refer to the local variable rather than the field. In this case, the local variable shadows the field in the method's scope. However, the method can use the this reference to refer to the shadowed field explicitly, as shown on the left sides of the assignments in lines 25–27 for SimpleTime's shadowed instance variables.

Method `buildString` (lines 31–36) returns a `String` created by a statement that uses the `this` reference explicitly and implicitly. Line 34 uses it explicitly to call method `toUniversalString`. Line 35 uses it implicitly to call the same method. Both lines perform the same task. You typically will not use `this` explicitly to reference other methods within the current object. Also, line 45 in method `toUniversalString` explicitly uses the `this` reference to access each instance variable. This is *not* necessary here, because the method does *not* have any local variables that shadow the instance variables of the class.

Common Programming Error F.1

It's often a logic error when a method contains a parameter or local variable that has the same name as a field of the class. In this case, use reference `this` if you wish to access the field of the class—otherwise, the method parameter or local variable will be referenced.

Error-Prevention Tip F.1

Avoid method-parameter names or local-variable names that conflict with field names. This helps prevent subtle, hard-to-locate bugs.

Performance Tip F.1

Java conserves storage by maintaining only one copy of each method per class—this method is invoked by every *object of the class. Each object, on the other hand, has its own copy of the class's instance variables (i.e., non-`static` fields). Each method of the class implicitly uses `this` to determine the specific object of the class to manipulate.*

Application class `ThisTest` (lines 4–11) demonstrates class `SimpleTime`. Line 8 creates an instance of class `SimpleTime` and invokes its constructor. Line 9 invokes the object's `buildString` method, then displays the results.

F.5 `Time` Class Case Study: Overloaded Constructors

As you know, you can declare your own constructor to specify how objects of a class should be initialized. Next, we demonstrate a class with several **overloaded constructors** that enable objects of that class to be initialized in different ways. To overload constructors, simply provide multiple constructor declarations with different signatures.

Class `Time2` with Overloaded Constructors

The default constructor for class `Time1` (Fig. F.1) initialized `hour`, `minute` and `second` to their default `0` values (which is midnight in universal time). The default constructor does not enable the class's clients to initialize the time with specific nonzero values. Class `Time2` (Fig. F.5) contains five overloaded constructors that provide convenient ways to initialize objects of the new class `Time2`. Each constructor initializes the object to begin in a consistent state. In this program, four of the constructors invoke a fifth, which in turn calls method `setTime` to ensure that the value supplied for `hour` is in the range 0 to 23, and the values for `minute` and `second` are each in the range 0 to 59. The compiler invokes the appropriate constructor by matching the number, types and order of the types of the arguments specified in the constructor call with the number, types and order of the types of the parameters specified in each constructor declaration. Class `Time2` also provides *set* and *get* methods for each instance variable.

```java
// Fig. F.5: Time2.java
// Time2 class with overloaded constructors.

public class Time2
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // Time2 no-argument constructor:
   // initializes each instance variable to zero
   public Time2()
   {
      this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 no-argument constructor

   // Time2 constructor: hour supplied, minute and second defaulted to 0
   public Time2( int h )
   {
      this( h, 0, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 one-argument constructor

   // Time2 constructor: hour and minute supplied, second defaulted to 0
   public Time2( int h, int m )
   {
      this( h, m, 0 ); // invoke Time2 constructor with three arguments
   } // end Time2 two-argument constructor

   // Time2 constructor: hour, minute and second supplied
   public Time2( int h, int m, int s )
   {
      setTime( h, m, s ); // invoke setTime to validate time
   } // end Time2 three-argument constructor

   // Time2 constructor: another Time2 object supplied
   public Time2( Time2 time )
   {
      // invoke Time2 three-argument constructor
      this( time.getHour(), time.getMinute(), time.getSecond() );
   } // end Time2 constructor with a Time2 object argument

   // Set Methods
   // set a new time value using universal time;
   // validate the data
   public void setTime( int h, int m, int s )
   {
      setHour( h ); // set the hour
      setMinute( m ); // set the minute
      setSecond( s ); // set the second
   } // end method setTime

```

Fig. F.5 | Time2 class with overloaded constructors. (Part 1 of 3.)

```
   // validate and set hour
   public void setHour( int h )
   {
      if ( h >= 0 && h < 24 )
         hour = h;
      else
         throw new IllegalArgumentException( "hour must be 0-23" );
   } // end method setHour

   // validate and set minute
   public void setMinute( int m )
   {
      if ( m >= 0 && m < 60 )
         minute = m;
      else
         throw new IllegalArgumentException( "minute must be 0-59" );
   } // end method setMinute

   // validate and set second
   public void setSecond( int s )
   {
      if ( s >= 0 && s < 60 )
         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
      else
         throw new IllegalArgumentException( "second must be 0-59" );
   } // end method setSecond

   // Get Methods
   // get hour value
   public int getHour()
   {
      return hour;
   } // end method getHour

   // get minute value
   public int getMinute()
   {
      return minute;
   } // end method getMinute

   // get second value
   public int getSecond()
   {
      return second;
   } // end method getSecond

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      return String.format(
         "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
   } // end method toUniversalString

```

Fig. F.5 | Time2 class with overloaded constructors. (Part 2 of 3.)

```
105     // convert to String in standard-time format (H:MM:SS AM or PM)
106     public String toString()
107     {
108        return String.format( "%d:%02d:%02d %s",
109           ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
110           getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
111     } // end method toString
112  } // end class Time2
```

Fig. F.5 | `Time2` class with overloaded constructors. (Part 3 of 3.)

Class `Time2`'s Constructors

Lines 12–15 declare a so-called **no-argument constructor** that's invoked without arguments. Once you declare any constructors in a class, the compiler will *not* provide a default constructor. This no-argument constructor ensures that class `Time2`'s clients can create `Time2` objects with default values. Such a constructor simply initializes the object as specified in the constructor's body. In the body, we introduce a use of the `this` reference that's allowed only as the *first* statement in a constructor's body. Line 14 uses `this` in method-call syntax to invoke the `Time2` constructor that takes three parameters (lines 30–33) with values of 0 for the `hour`, `minute` and `second`. Using the `this` reference as shown here is a popular way to reuse initialization code provided by another of the class's constructors rather than defining similar code in the no-argument constructor's body. We use this syntax in four of the five `Time2` constructors to make the class easier to maintain and modify. If we need to change how objects of class `Time2` are initialized, only the constructor that the class's other constructors call will need to be modified. In fact, even that constructor might not need modification in this example. That constructor simply calls the `setTime` method to perform the actual initialization, so it's possible that the changes the class might require would be localized to the *set* methods.

Common Programming Error F.2

It's a compilation error when `this` is used in a constructor's body to call another constructor of the same class if that call is not the first *statement in the constructor. It's also a compilation error when a method attempts to invoke a constructor directly via `this`.*

Common Programming Error F.3

A constructor can call methods of the class. Be aware that the instance variables might not yet be initialized, because the constructor is in the process of initializing the object. Using instance variables before they've been initialized properly is a logic error.

Lines 18–21 declare a `Time2` constructor with a single `int` parameter representing the `hour`, which is passed with 0 for the `minute` and `second` to the constructor at lines 30–33. Lines 24–27 declare a `Time2` constructor that receives two `int` parameters representing the `hour` and `minute`, which are passed with 0 for the `second` to the constructor at lines 30–33. Like the no-argument constructor, each of these constructors invokes the constructor at lines 30–33 to minimize code duplication. Lines 30–33 declare the `Time2` constructor that receives three `int` parameters representing the `hour`, `minute` and `second`. This constructor calls `setTime` to initialize the instance variables.

Lines 36–40 declare a `Time2` constructor that receives a reference to another `Time2` object. In this case, the values from the `Time2` argument are passed to the three-argument constructor at lines 30–33 to initialize the `hour`, `minute` and `second`. Line 39 could have directly accessed the `hour`, `minute` and `second` values of the constructor's argument `time` with the expressions `time.hour`, `time.minute` and `time.second`—even though `hour`, `minute` and `second` are declared as `private` variables of class `Time2`. This is due to a special relationship between objects of the same class. We'll see in a moment why it's preferable to use the *get* methods.

Software Engineering Observation F.1

When one object of a class has a reference to another object of the same class, the first object can access all *the second object's data and methods (including those that are* `private`*).*

Class `Time2`'s `setTime` Method

Method `setTime` (lines 45–50) invokes the `setHour` (lines 53–59), `setMinute` (lines 62–68) and `setSecond` (lines 71–77) methods, which ensure that the value supplied for `hour` is in the range 0 to 23 and the values for `minute` and `second` are each in the range 0 to 59. If a value is out of range, each of these methods throws an `IllegalArgumentException` (lines 58, 67 and 76) indicating which value was out of range.

Notes Regarding Class `Time2`'s set *and* get *Methods and Constructors*

`Time2`'s *set* and *get* methods are called throughout the class. In particular, method `setTime` calls methods `setHour`, `setMinute` and `setSecond` in lines 47–49, and methods `toUniversalString` and `toString` call methods `getHour`, `getMinute` and `getSecond` in line 102 and lines 109–110, respectively. In each case, these methods could have accessed the class's private data directly without calling the *set* and *get* methods. However, consider changing the representation of the time from three `int` values (requiring 12 bytes of memory) to a single `int` value representing the total number of seconds that have elapsed since midnight (requiring only 4 bytes of memory). If we made such a change, only the bodies of the methods that access the `private` data directly would need to change—in particular, the individual *set* and *get* methods for the `hour`, `minute` and `second`. There would be no need to modify the bodies of methods `setTime`, `toUniversalString` or `toString` because they do not access the data directly. Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.

Similarly, each `Time2` constructor could include a copy of the appropriate statements from methods `setHour`, `setMinute` and `setSecond`. Doing so may be slightly more efficient, because the extra calls to the constructor and `setTime` are eliminated. However, *duplicating* statements in multiple methods or constructors makes changing the class's internal data representation more difficult. Having the `Time2` constructors call the constructor with three arguments (or even call `setTime` directly) requires that any changes to the implementation of `setTime` be made only once. Also, the compiler can optimize programs by removing calls to simple methods and replacing them with the expanded code of their declarations—a technique known as **inlining the code**, which improves program performance.

Software Engineering Observation F.2

When implementing a method of a class, use the class's set *and* get *methods to access the class's* `private` *data. This simplifies code maintenance and reduces the likelihood of errors.*

Using Class `Time2`'s Overloaded Constructors

Class `Time2Test` (Fig. F.6) invokes the overloaded `Time2` constructors (lines 8–12 and 40). Line 8 invokes the no-argument constructor (Fig. F.5, lines 12–15). Lines 9–13 of the program demonstrate passing arguments to the other `Time2` constructors. Line 9 invokes the single-argument constructor that receives an `int` at lines 18–21 of Fig. F.5. Line 10 invokes the two-argument constructor at lines 24–27 of Fig. F.5. Line 11 invokes the three-argument constructor at lines 30–33 of Fig. F.5. Line 12 invokes the single-argument constructor that takes a `Time2` at lines 36–40 of Fig. F.5. Next, the application displays the `String` representations of each `Time2` object to confirm that it was initialized properly. Line 40 attempts to intialize `t6` by creating a new `Time2` object and passing three invalid values to the constructor. When the constructor attempts to use the invalid hour value to initialize the object's `hour`, an `IllegalArgumentException` occurs. We catch this exception at line 42 and display its error message, which results in the last line of the output.

```
 1  // Fig. F.6: Time2Test.java
 2  // Overloaded constructors used to initialize Time2 objects.
 3
 4  public class Time2Test
 5  {
 6     public static void main( String[] args )
 7     {
 8        Time2 t1 = new Time2(); // 00:00:00
 9        Time2 t2 = new Time2( 2 ); // 02:00:00
10        Time2 t3 = new Time2( 21, 34 ); // 21:34:00
11        Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12        Time2 t5 = new Time2( t4 ); // 12:25:42
13
14        System.out.println( "Constructed with:" );
15        System.out.println( "t1: all arguments defaulted" );
16        System.out.printf( "   %s\n", t1.toUniversalString() );
17        System.out.printf( "   %s\n", t1.toString() );
18
19        System.out.println(
20           "t2: hour specified; minute and second defaulted" );
21        System.out.printf( "   %s\n", t2.toUniversalString() );
22        System.out.printf( "   %s\n", t2.toString() );
23
24        System.out.println(
25           "t3: hour and minute specified; second defaulted" );
26        System.out.printf( "   %s\n", t3.toUniversalString() );
27        System.out.printf( "   %s\n", t3.toString() );
28
29        System.out.println( "t4: hour, minute and second specified" );
30        System.out.printf( "   %s\n", t4.toUniversalString() );
31        System.out.printf( "   %s\n", t4.toString() );
32
33        System.out.println( "t5: Time2 object t4 specified" );
34        System.out.printf( "   %s\n", t5.toUniversalString() );
35        System.out.printf( "   %s\n", t5.toString() );
36
```

Fig. F.6 | Overloaded constructors used to initialize `Time2` objects. (Part 1 of 2.)

```
      // attempt to initialize t6 with invalid values
      try
      {
         Time2 t6 = new Time2( 27, 74, 99 ); // invalid values
      } // end try
      catch ( IllegalArgumentException e )
      {
         System.out.printf( "\nException while initializing t6: %s\n",
            e.getMessage() );
      } // end catch
   } // end main
} // end class Time2Test
```

```
Constructed with:
t1: all arguments defaulted
   00:00:00
   12:00:00 AM
t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM
t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM

Exception while initializing t6: hour must be 0-23
```

Fig. F.6 | Overloaded constructors used to initialize `Time2` objects. (Part 2 of 2.)

F.6 Default and No-Argument Constructors

Every class must have at least one constructor. If you do not provide any in a class's declaration, the compiler creates a default constructor that takes no arguments when it's invoked. The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values (zero for primitive numeric types, `false` for `boolean` values and `null` for references). In Section G.4.1, you'll learn that the default constructor performs another task also.

If your class declares constructors, the compiler will *not* create a default constructor. In this case, you must declare a no-argument constructor if default initialization is required. Like a default constructor, a no-argument constructor is invoked with empty parentheses. The `Time2` no-argument constructor (lines 12–15 of Fig. F.5) explicitly initializes a `Time2` object by passing to the three-argument constructor 0 for each parameter. Since 0 is the default value for `int` instance variables, the no-argument constructor in this example could actually be declared with an empty body. In this case, each instance variable would receive its default value when the no-argument constructor was called. If we omit the no-argument constructor, clients of this class would not be able to create a `Time2` object with the expression `new Time2()`.

F.7 Composition

A class can have references to objects of other classes as members. This is called **composition** and is sometimes referred to as a ***has-a*** **relationship**. For example, an `AlarmClock` object needs to know the current time *and* the time when it's supposed to sound its alarm, so it's reasonable to include *two* references to `Time` objects in an `AlarmClock` object.

Class `Date`

This composition example contains classes `Date` (Fig. F.7), `Employee` (Fig. F.8) and `EmployeeTest` (Fig. F.9). Class `Date` (Fig. F.7) declares instance variables month, day and year (lines 6–8) to represent a date. The constructor receives three `int` parameters. Line 17 invokes utility method `checkMonth` (lines 26–32) to validate the month—if the value is out of range the method throws an exception. Line 15 assumes that the value for `year` is correct and doesn't validate it. Line 19 invokes utility method `checkDay` (lines 35–48) to validate the `day` based on the current `month` and `year`. Line 38 determines whether the day is correct based on the number of days in the particular `month`. If the day is not correct, lines 42–43 determine whether the `month` is February, the day is `29` and the `year` is a leap year. If the `day` is still invalid, the method throws an exception. Lines 21–22 in the constructor output the `this` reference as a `String`. Since `this` is a reference to the current `Date` object, the object's `toString` method (lines 51–54) is called *implicitly* to obtain the object's `String` representation.

```
 1  // Fig. F.7: Date.java
 2  // Date class declaration.
 3
 4  public class Date
 5  {
 6     private int month; // 1-12
 7     private int day; // 1-31 based on month
 8     private int year; // any year
 9
10     private static final int[] daysPerMonth = // days in each month
11        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13     // constructor: call checkMonth to confirm proper value for month;
14     // call checkDay to confirm proper value for day
15     public Date( int theMonth, int theDay, int theYear )
16     {
17        month = checkMonth( theMonth ); // validate month
18        year = theYear; // could validate year
19        day = checkDay( theDay ); // validate day
20
21        System.out.printf(
22           "Date object constructor for date %s\n", this );
23     } // end Date constructor
24
25     // utility method to confirm proper month value
26     private int checkMonth( int testMonth )
27     {
28        if ( testMonth > 0 && testMonth <= 12 ) // validate month
29           return testMonth;
```

Fig. F.7 | `Date` class declaration. (Part 1 of 2.)

```java
      else // month is invalid
         throw new IllegalArgumentException( "month must be 1-12" );
   } // end method checkMonth

   // utility method to confirm proper day value based on month and year
   private int checkDay( int testDay )
   {
      // check if day in range for month
      if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
         return testDay;

      // check for leap year
      if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
           ( year % 4 == 0 && year % 100 != 0 ) ) )
         return testDay;

      throw new IllegalArgumentException(
         "day out-of-range for the specified month and year" );
   } // end method checkDay

   // return a String of the form month/day/year
   public String toString()
   {
      return String.format( "%d/%d/%d", month, day, year );
   } // end method toString
} // end class Date
```

Fig. F.7 | `Date` class declaration. (Part 2 of 2.)

Class `Employee`

Class `Employee` (Fig. F.8) has instance variables `firstName`, `lastName`, `birthDate` and `hireDate`. Members `firstName` and `lastName` (lines 6–7) are references to `String` objects. Members `birthDate` and `hireDate` (lines 8–9) are references to `Date` objects. This demonstrates that a class can have as instance variables references to objects of other classes. The `Employee` constructor (lines 12–19) takes four parameters—`first`, `last`, `dateOfBirth` and `dateOfHire`. The objects referenced by the parameters are assigned to the `Employee` object's instance variables. When class `Employee`'s `toString` method is called, it returns a `String` containing the employee's name and the `String` representations of the two `Date` objects. Each of these `Strings` is obtained with an *implicit* call to the `Date` class's `toString` method.

```java
1  // Fig. F.8: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
```

Fig. F.8 | `Employee` class with references to other objects. (Part 1 of 2.)

```

   // constructor to initialize name, birth date and hire date
   public Employee( String first, String last, Date dateOfBirth,
      Date dateOfHire )
   {
      firstName = first;
      lastName = last;
      birthDate = dateOfBirth;
      hireDate = dateOfHire;
   } // end Employee constructor

   // convert Employee to String format
   public String toString()
   {
      return String.format( "%s, %s  Hired: %s  Birthday: %s",
         lastName, firstName, hireDate, birthDate );
   } // end method toString
} // end class Employee
```

Fig. F.8 | `Employee` class with references to other objects. (Part 2 of 2.)

Class `EmployeeTest`

Class `EmployeeTest` (Fig. F.9) creates two `Date` objects (lines 8–9) to represent an `Employee`'s birthday and hire date, respectively. Line 10 creates an `Employee` and initializes its instance variables by passing to the constructor two `String`s (representing the `Employee`'s first and last names) and two `Date` objects (representing the birthday and hire date). Line 12 implicitly invokes the `Employee`'s `toString` method to display the values of its instance variables and demonstrate that the object was initialized properly.

```
 1  // Fig. F.9: EmployeeTest.java
 2  // Composition demonstration.
 3
 4  public class EmployeeTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        Date birth = new Date( 7, 24, 1949 );
 9        Date hire = new Date( 3, 12, 1988 );
10        Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12        System.out.println( employee );
13     } // end main
14  } // end class EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Fig. F.9 | Composition demonstration.

F.8 Enumerations

In Fig. D.5, we introduced the basic enum type, which defines a set of constants represented as unique identifiers. In that program the enum constants represented the game's status. In this section we discuss the relationship between enum types and classes. Like classes, all enum types are reference types. An enum type is declared with an **enum declaration**, which is a comma-separated list of enum constants—the declaration may optionally include other components of traditional classes, such as constructors, fields and methods. Each enum declaration declares an enum class with the following restrictions:

1. enum constants are implicitly final, because they declare constants that shouldn't be modified.
2. enum constants are implicitly static.
3. Any attempt to create an object of an enum type with operator new results in a compilation error.

The enum constants can be used anywhere constants can be used, such as in the case labels of switch statements and to control enhanced for statements.

Figure F.10 illustrates how to declare instance variables, a constructor and methods in an enum type. The enum declaration (lines 5–37) contains two parts—the enum constants and the other members of the enum type. The first part (lines 8–13) declares six enum constants. Each is optionally followed by arguments which are passed to the **enum constructor** (lines 20–24). Like the constructors you've seen in classes, an enum constructor can specify any number of parameters and can be overloaded. In this example, the enum constructor requires two String parameters. To properly initialize each enum constant, we follow it with parentheses containing two String arguments, which are passed to the enum's constructor. The second part (lines 16–36) declares the other members of the enum type—two instance variables (lines 16–17), a constructor (lines 20–24) and two methods (lines 27–30 and 33–36).

```
 1  // Fig. F.10: Book.java
 2  // Declaring an enum type with constructor and explicit instance fields
 3  // and accessors for these fields
 4
 5  public enum Book
 6  {
 7     // declare constants of enum type
 8     JHTP( "Java How to Program", "2012" ),
 9     CHTP( "C How to Program", "2007" ),
10     IW3HTP( "Internet & World Wide Web How to Program", "2008" ),
11     CPPHTP( "C++ How to Program", "2012" ),
12     VBHTP( "Visual Basic 2010 How to Program", "2011" ),
13     CSHARPHTP( "Visual C# 2010 How to Program", "2011" );
14
15     // instance fields
16     private final String title; // book title
17     private final String copyrightYear; // copyright year
```

Fig. F.10 | Declaring an enum type with constructor and explicit instance fields and accessors for these fields. (Part 1 of 2.)

```java
18
19     // enum constructor
20     Book( String bookTitle, String year )
21     {
22        title = bookTitle;
23        copyrightYear = year;
24     } // end enum Book constructor
25
26     // accessor for field title
27     public String getTitle()
28     {
29        return title;
30     } // end method getTitle
31
32     // accessor for field copyrightYear
33     public String getCopyrightYear()
34     {
35        return copyrightYear;
36     } // end method getCopyrightYear
37  } // end enum Book
```

Fig. F.10 | Declaring an `enum` type with constructor and explicit instance fields and accessors for these fields. (Part 2 of 2.)

Lines 16–17 declare the instance variables `title` and `copyrightYear`. Each `enum` constant in `Book` is actually an object of type `Book` that has its own copy of instance variables `title` and `copyrightYear`. The constructor (lines 20–24) takes two `String` parameters, one that specifies the book's title and one that specifies its copyright year. Lines 22–23 assign these parameters to the instance variables. Lines 27–36 declare two methods, which return the book title and copyright year, respectively.

Figure F.11 tests the `enum` type `Book` and illustrates how to iterate through a range of `enum` constants. For every `enum`, the compiler generates the `static` method **`values`** (called in line 12) that returns an array of the `enum`'s constants in the order they were declared. Lines 12–14 use the enhanced `for` statement to display all the constants declared in the `enum Book`. Line 14 invokes the `enum Book`'s `getTitle` and `getCopyrightYear` methods to get the title and copyright year associated with the constant. When an `enum` constant is converted to a `String` (e.g., `book` in line 13), the constant's identifier is used as the `String` representation (e.g., `JHTP` for the first `enum` constant).

```java
 1  // Fig. F.11: EnumTest.java
 2  // Testing enum type Book.
 3  import java.util.EnumSet;
 4
 5  public class EnumTest
 6  {
 7     public static void main( String[] args )
 8     {
 9        System.out.println( "All books:\n" );
```

Fig. F.11 | Testing `enum` type `Book`. (Part 1 of 2.)

```
10
11         // print all books in enum Book
12         for ( Book book : Book.values() )
13            System.out.printf( "%-10s%-45s%s\n", book,
14                book.getTitle(), book.getCopyrightYear() );
15
16         System.out.println( "\nDisplay a range of enum constants:\n" );
17
18         // print first four books
19         for ( Book book : EnumSet.range( Book.JHTP, Book.CPPHTP ) )
20            System.out.printf( "%-10s%-45s%s\n", book,
21                book.getTitle(), book.getCopyrightYear() );
22      } // end main
23   } // end class EnumTest
```

```
All books:

JHTP      Java How to Program                          2012
CHTP      C How to Program                             2007
IW3HTP    Internet & World Wide Web How to Program     2008
CPPHTP    C++ How to Program                           2012
VBHTP     Visual Basic 2010 How to Program             2011
CSHARPHTP Visual C# 2010 How to Program                2011

Display a range of enum constants:

JHTP      Java How to Program                          2012
CHTP      C How to Program                             2007
IW3HTP    Internet & World Wide Web How to Program     2008
CPPHTP    C++ How to Program                           2012
```

Fig. F.11 | Testing `enum` type `Book`. (Part 2 of 2.)

Lines 19–21 use the `static` method **`range`** of class **`EnumSet`** (declared in package `java.util`) to display a range of the `enum Book`'s constants. Method `range` takes two parameters—the first and the last `enum` constants in the range—and returns an `EnumSet` that contains all the constants between these two constants, inclusive. For example, the expression `EnumSet.range( Book.JHTP, Book.CPPHTP )` returns an `EnumSet` containing `Book.JHTP`, `Book.CHTP`, `Book.IW3HTP` and `Book.CPPHTP`. The enhanced `for` statement can be used with an `EnumSet` just as it can with an array, so lines 12–14 use it to display the title and copyright year of every book in the `EnumSet`. Class `EnumSet` provides several other `static` methods for creating sets of `enum` constants from the same `enum` type.

Common Programming Error F.4

In an `enum` declaration, it's a syntax error to declare `enum` constants after the `enum` type's constructors, fields and methods.

F.9 Garbage Collection

Every object uses system resources, such as memory. We need a disciplined way to give resources back to the system when they're no longer needed; otherwise, "resource leaks" might occur that would prevent them from being reused by your program or possibly by other programs. The JVM performs automatic **garbage collection** to reclaim the memory occupied by objects that are no longer used. When there are no more references to an object, the object is eligible to be collected. This typically occurs when the JVM executes its **garbage collector**. So, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways. Other types of resource leaks can occur. For example, an application may open a file on disk to modify its contents. If it does not close the file, the application must terminate before any other application can use it.

Software Engineering Observation F.3

A class that uses system resources, such as files on disk, should provide a method that programmers can call to release resources when they're no longer needed in a program. Many Java API classes provide `close` or `dispose` methods for this purpose. For example, class `Scanner` has a `close` method.

F.10 static Class Members

Every object has its own copy of all the instance variables of the class. In certain cases, only one copy of a particular variable should be *shared* by all objects of a class. A **`static` field**—called a **class variable**—is used in such cases. A `static` variable represents **classwide information**—all objects of the class share the *same* piece of data. The declaration of a `static` variable begins with the keyword `static`.

Let's motivate `static` data with an example. Suppose that we have a video game with `Martian`s and other space creatures. Each `Martian` tends to be brave and willing to attack other space creatures when the `Martian` is aware that at least four other `Martian`s are present. If fewer than five `Martian`s are present, each of them becomes cowardly. Thus, each `Martian` needs to know the `martianCount`. We could endow class `Martian` with `martianCount` as an instance variable. If we do this, then every `Martian` will have *a separate copy* of the instance variable, and every time we create a new `Martian`, we'll have to update the instance variable `martianCount` in every `Martian` object. This wastes space with the redundant copies, wastes time in updating the separate copies and is error prone. Instead, we declare `martianCount` to be `static`, making `martianCount` classwide data. Every `Martian` can see the `martianCount` as if it were an instance variable of class `Martian`, but only one copy of the `static` `martianCount` is maintained. This saves space. We save time by having the `Martian` constructor increment the `static` `martianCount`—there's only one copy, so we do not have to increment separate copies for each `Martian` object.

Software Engineering Observation F.4

Use a `static` variable when all objects of a class must use the same copy of the variable.

Static variables have class scope. We can access a class's `public static` members through a reference to any object of the class, or by qualifying the member name with the class name and a dot (`.`), as in `Math.random()`. A class's `private static` class members can be accessed by client code only through methods of the class. Actually, *`static` class members exist even when no objects of the class exist*—they're available as soon as the class is loaded into memory at execution time. To access a `public static` member when no objects of the class exist (and even when they do), prefix the class name and a dot (`.`) to the `static` member, as in `Math.PI`. To access a `private static` member when no objects of the class exist, provide a `public static` method and call it by qualifying its name with the class name and a dot.

Software Engineering Observation F.5

Static class variables and methods exist, and can be used, even if no objects of that class have been instantiated.

A `static` method cannot access non-`static` class members, because a `static` method can be called even when no objects of the class have been instantiated. For the same reason, the `this` reference cannot be used in a `static` method. The `this` reference must refer to a specific object of the class, and when a `static` method is called, there might not be any objects of its class in memory.

Common Programming Error F.5

A compilation error occurs if a `static` method calls an instance (non-`static`) method in the same class by using only the method name. Similarly, a compilation error occurs if a `static` method attempts to access an instance variable in the same class by using only the variable name.

Common Programming Error F.6

Referring to `this` in a `static` method is a compilation error.

Tracking the Number of Employee Objects That Have Been Created

Our next program declares two classes—`Employee` (Fig. F.12) and `EmployeeTest` (Fig. F.13). Class `Employee` declares `private static` variable `count` (Fig. F.12, line 9) and `public static` method `getCount` (lines 36–39). The `static` variable `count` is initialized to zero in line 9. If a `static` variable is not initialized, the compiler assigns it a default value—in this case `0`, the default value for type `int`. Variable `count` maintains a count of the number of objects of class `Employee` that have been created so far.

```
 1  // Fig. F.12: Employee.java
 2  // Static variable used to maintain a count of the number of
 3  // Employee objects in memory.
 4
 5  public class Employee
 6  {
 7     private String firstName;
 8     private String lastName;
 9     private static int count = 0; // number of Employees created
10
11     // initialize Employee, add 1 to static count and
12     // output String indicating that constructor was called
13     public Employee( String first, String last )
14     {
```

Fig. F.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 1 of 2.)

```
15         firstName = first;
16         lastName = last;
17
18         ++count;  // increment static count of employees
19         System.out.printf( "Employee constructor: %s %s; count = %d\n",
20            firstName, lastName, count );
21      } // end Employee constructor
22
23      // get first name
24      public String getFirstName()
25      {
26         return firstName;
27      } // end method getFirstName
28
29      // get last name
30      public String getLastName()
31      {
32         return lastName;
33      } // end method getLastName
34
35      // static method to get static count value
36      public static int getCount()
37      {
38         return count;
39      } // end method getCount
40   } // end class Employee
```

Fig. F.12 | `static` variable used to maintain a count of the number of `Employee` objects in memory. (Part 2 of 2.)

When `Employee` objects exist, variable `count` can be used in any method of an `Employee` object—this example increments `count` in the constructor (line 18). The `public static` method `getCount` (lines 36–39) returns the number of `Employee` objects that have been created so far. When no objects of class `Employee` exist, client code can access variable `count` by calling method `getCount` via the class name, as in `Employee.getCount()`. When objects exist, method `getCount` can also be called via any reference to an `Employee` object.

Good Programming Practice F.1

Invoke every `static` method by using the class name and a dot (`.`) to emphasize that the method being called is a `static` method.

`EmployeeTest` method `main` (Fig. F.13) instantiates two `Employee` objects (lines 13–14). When each `Employee` object's constructor is invoked, lines 15–16 of Fig. F.12 assign the `Employee`'s first name and last name to instance variables `firstName` and `lastName`. These two statements do *not* make copies of the original `String` arguments. Actually, `String` objects in Java are **immutable**—they cannot be modified after they're created. Therefore, it's safe to have many references to one `String` object. This is not normally the case for objects of most other classes in Java. If `String` objects are immutable, you might wonder why we're able to use operators + and += to concatenate `String` objects. String-concatenation operations actually result in a *new* `Strings` object containing the concatenated values. The original `String` objects are not modified.

When `main` has finished using the two `Employee` objects, the references `e1` and `e2` are set to `null` at lines 31–32 (Fig. F.13). At this point, references `e1` and `e2` no longer refer to the objects that were instantiated in lines 13–14. The objects become "eligible for garbage collection" because there are no more references to them in the program.

```
 1  // Fig. F.13: EmployeeTest.java
 2  // static member demonstration.
 3
 4  public class EmployeeTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // show that count is 0 before creating Employees
 9        System.out.printf( "Employees before instantiation: %d\n",
10           Employee.getCount() );
11
12        // create two Employees; count should be 2
13        Employee e1 = new Employee( "Susan", "Baker" );
14        Employee e2 = new Employee( "Bob", "Blue" );
15
16        // show that count is 2 after creating two Employees
17        System.out.println( "\nEmployees after instantiation: " );
18        System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19        System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20        System.out.printf( "via Employee.getCount(): %d\n",
21           Employee.getCount() );
22
23        // get names of Employees
24        System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
25           e1.getFirstName(), e1.getLastName(),
26           e2.getFirstName(), e2.getLastName() );
27
28        // in this example, there is only one reference to each Employee,
29        // so the following two statements indicate that these objects
30        // are eligible for garbage collection
31        e1 = null;
32        e2 = null;
33     } // end main
34  } // end class EmployeeTest
```

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

Fig. F.13 | `static` member demonstration.

Eventually, the garbage collector might reclaim the memory for these objects (or the operating system will reclaim the memory when the program terminates). The JVM does not guarantee when, or even whether, the garbage collector will execute. When it does, it's possible that no objects or only a subset of the eligible objects will be collected.

F.11 `final` Instance Variables

The **principle of least privilege** is fundamental to good software engineering. In the context of an application, it states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more. This makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible.

Let's see how this principle applies to instance variables. Some of them need to be modifiable and some do not. You can use the keyword `final` to specify that a variable is not modifiable (i.e., it's a constant) and that any attempt to modify it is an error. For example,

```
    private final int INCREMENT;
```

declares a `final` (constant) instance variable `INCREMENT` of type `int`. Such variables can be initialized when they're declared. If they are not, they *must* be initialized in every constructor of the class. Initializing constants in constructors enables each object of the class to have a different value for the constant. If a `final` variable is not initialized in its declaration or in every constructor, a compilation error occurs.

Software Engineering Observation F.6

Declaring an instance variable as `final` helps enforce the principle of least privilege. If an instance variable should not be modified, declare it to be `final` to prevent modification.

Common Programming Error F.7

Attempting to modify a `final` instance variable after it's initialized is a compilation error.

Error-Prevention Tip F.2

Attempts to modify a `final` instance variable are caught at compilation time rather than causing execution-time errors. It's always preferable to get bugs out at compilation time, if possible, rather than allow them to slip through to execution time (where experience has found that repair is often many times more expensive).

Software Engineering Observation F.7

A `final` field should also be declared `static` if it's initialized in its declaration to a value that's the same for all objects of the class. After this initialization, its value can never change. Therefore, we don't need a separate copy of the field for every object of the class. Making the field `static` enables all objects of the class to share the `final` field.

F.12 Packages

We've seen in almost every example in the text that classes from preexisting libraries, such as the Java API, can be imported into a Java program. Each class in the Java API belongs to a package that contains a group of related classes. These packages are defined once, but can be imported into many programs. As applications become more complex, packages help you manage the complexity of application components. Packages also facilitate software reuse by enabling programs to *import* classes from other packages (as we've done in most examples), rather than *copying* the classes into each program that uses them. Another benefit of packages is that they provide a convention for unique class names, which helps prevent class-name conflicts.

F.13 Package Access

If no access modifier (`public`, `protected` or `private`) is specified for a method or variable when it's declared in a class, the method or variable has **package access**. In a program that consists of one class declaration, this has no specific effect. However, if a program uses multiple classes from the same package (i.e., a group of related classes), these classes can access each other's package-access members directly through references to objects of the appropriate classes, or in the case of `static` members through the class name. Package access is rarely used.

F.14 Wrap-Up

In this appendix, we presented additional class concepts. The `Time` class case study presented a complete class declaration consisting of `private` data, overloaded `public` constructors for initialization flexibility, *set* and *get* methods for manipulating the class's data, and methods that returned `String` representations of a `Time` object in two different formats. You also learned that every class can declare a `toString` method that returns a `String` representation of an object of the class and that method `toString` can be called implicitly whenever an object of a class appears in the code where a `String` is expected.

You learned that the `this` reference is used implicitly in a class's non-`static` methods to access the class's instance variables and other non-`static` methods. You also saw explicit uses of the `this` reference to access the class's members (including shadowed fields) and how to use keyword `this` in a constructor to call another constructor of the class.

We discussed the differences between default constructors provided by the compiler and no-argument constructors provided by the programmer. You learned that a class can have references to objects of other classes as members—a concept known as composition. You saw the `enum` class type and learned how it can be used to create a set of constants for use in a program. You learned about Java's garbage-collection capability and how it (unpredictably) reclaims the memory of objects that are no longer used. We explained the motivation for `static` fields in a class and demonstrated how to declare and use `static` fields and methods in your own classes. You also learned how to declare and initialize `final` variables.

You learned that fields declared without an access modifier are given package access by default and that classes in the same package can access the package-access members of other classes in the package.

In the next appendix, you'll learn about two important aspects of object-oriented programming in Java—inheritance and polymorphism. You'll see that all classes in Java are related directly or indirectly to the class called `Object`. You'll also begin to understand how the relationships between classes enable you to build more powerful applications.

Self-Review Exercise

F.1 Fill in the blanks in each of the following statements:

a) The `public` methods of a class are also known as the class's _______ or _______.
b) Method _______ takes no arguments and returns a `String` in universal-time format, consisting of two digits each for the hour, minute and second.
c) If a method contains a local variable with the same name as one of its class's fields, the local variable _______ the field in that method's scope.
d) Keyword _______ specifies that a variable is not modifiable.
e) The _______ states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task.
f) If a class declares constructors, the compiler will not create a(n) _______.
g) An object's _______ method is called implicitly when an object appears in code where a `String` is needed.
h) For every `enum`, the compiler generates a `static` method called _______ that returns an array of the `enum`'s constants in the order in which they were declared.
i) Composition is sometimes referred to as a(n) _______ relationship.
j) A(n) _______ declaration contains a comma-separated list of constants.
k) A(n) _______ variable represents classwide information that's shared by all the objects of the class.

Answers to Self-Review Exercise

F.1 a) `public` services, `public` interface. b) `toUniversalString`. c) shadows. d) `final`. e) principle of least privilege. f) default constructor. g) `toString`. h) `values`. i) *has-a*. j) `enum`. k) `static`.

Exercises

F.2 *(Cuboid Class)* Create a class `Cuboid` with attributes `length`, `width` and `breadth`, each of which defaults to 1. Provide methods that calculate the cuboid's area. Provide *set* and *get* methods for `length`, `width` and `breadth`. The *set* methods should verify that `length`, `width` and `breadth` are each floating-point numbers larger than 0.0 and less than 20.0. Write a program to test class `Cuboid`.

F.3 *(Savings Account Class)* Create class `SavingsAccount`. Use a `static` variable `annualInterestRate` to store the annual interest rate for all account holders. Each object of the class should contain a `private` instance variable `savingsBalance` indicating the amount the saver currently has on deposit. Provide method `calculateMonthlyInterest` to calculate the monthly interest by multiplying the `savingsBalance` by `annualInterestRate` divided by 12—this interest should be added to `savingsBalance`. Provide a `static` method `modifyInterestRate` that sets the `annualInterestRate` to a new value. Write a program to test class `SavingsAccount`. Instantiate two `savingsAccount` objects, `saver1` and `saver2`, with balances of $2000.00 and $3000.00, respectively. Set `annualInterestRate` to 4%, then calculate the monthly interest for each of 12 months and print the new balances for both savers. Next, set the `annualInterestRate` to 5%, calculate the next month's interest and print the new balances for both savers.

F.4 *(Enhancing Class `Time2`)* Modify class `Time2` of Fig. F.5 to include a `tick` method that increments the time stored in a `Time2` object by one second. Provide method `incrementMinute` to increment the minute by one and method `incrementHour` to increment the hour by one. Write a program that tests the `tick` method, the `incrementMinute` method and the `incrementHour` method to ensure that they work correctly. Be sure to test the following cases:

a) incrementing into the next minute,
b) incrementing into the next hour and
c) incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

F.5 Write an `enum` type `TrafficLight`, whose constants (`RED`, `GREEN`, `YELLOW`) take one parameter—the duration of the light. Write a program to test the `TrafficLight` `enum` so that it displays the `enum` constants and their durations.

F.6 *(Date Class)* Create class `Date` with the following capabilities:

a) Output the date in multiple formats, such as

```
MM/DD/YYYY
June 14, 1992
DDD YYYY
```

b) Use overloaded constructors to create `Date` objects initialized with dates of the formats in part (a). In the first case the constructor should receive three integer values. In the second case it should receive a `String` and two integer values. In the third case it should receive two integer values, the first of which represents the day number in the year. [*Hint:* To convert the `String` representation of the month to a numeric value, compare `String`s using the `equals` method. For example, if `s1` and `s2` are `String`s, the method call `s1.equals( s2 )` returns `true` if the `String`s are identical and otherwise returns `false`.]

F.7 *(Huge Integer Class)* Create a class `HugeInteger` which uses a 40-element array of digits to store integers as large as 40 digits each. Provide methods `parse`, `toString`, `add` and `subtract`. Method `parse` should receive a `String`, extract each digit using method `charAt` and place the integer equivalent of each digit into the integer array. For comparing `HugeInteger` objects, provide the following methods: `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` and `isLessThanOrEqualTo`. Each of these so-called *predicate methods* (that is, methods that test a condition and return `true` or `false`) returns `true` if the relationship holds between the two `HugeInteger` objects and returns `false` if the relationship does not hold. Provide a predicate method `isZero`. If you feel ambitious, also provide methods `multiply`, `divide` and `remainder`. [*Note:* Primitive `boolean` values can be output as the word "true" or the word "false" with format specifier `%b`.]

F.8 *(Tic-Tac-Toe)* Create a class `TicTacToe` that will enable you to write a program to play Tic-Tac-Toe. The class contains a private 3-by-3 two-dimensional array. Use an enumeration to represent the value in each cell of the array. The enumeration's constants should be named `X`, `O` and `EMPTY` (for a position that does not contain an `X` or an `O`). The constructor should initialize the board elements to `EMPTY`. Allow two human players. Wherever the first player moves, place an `X` in the specified square, and place an `O` wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won and whether it's a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional Tic-Tac-Toe on a 4-by-4-by-4 board [*Note:* This is an extremely challenging project!].

Object-Oriented Programming: Inheritance and Polymorphism

G

Objectives

In this appendix you'll:

- Learn how inheritance promotes software resuse.
- Understand the relationships between superclasses and subclasses.
- Use keyword `extends` to effect inheritance.
- Use `protected` to give subclass methods access to superclass members.
- Reference superclass members with `super`.
- Learn the methods of class `Object`.
- Learn the concept of polymorphism.
- Use overridden methods to effect polymorphism.
- Distinguish between abstract and concrete classes.
- Declare abstract methods to create abstract classes.
- Learn how polymorphism makes systems extensible and maintainable.
- Determine an object's type at execution time.
- Declare and implement interfaces.

Outline

G.1 Introduction to Inheritance

The first part of this appendix continues our discussion of object-oriented programming (OOP) by introducing one of its primary capabilities—**inheritance**, which is a form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities. With inheritance, you can save time during program development by basing new classes on existing proven and debugged high-quality software. The existing class is called the **superclass**, and the new class is the **subclass.** Each subclass can become a superclass for future subclasses.

A subclass can add its own fields and methods. Therefore, a subclass is *more specific* than its superclass and represents a more specialized group of objects. The subclass exhibits the behaviors of its superclass and can modify those behaviors so that they operate appropriately for the subclass. This is why inheritance is sometimes referred to as **specialization**.

The **direct superclass** is the superclass from which the subclass explicitly inherits. An **indirect superclass** is any class above the direct superclass in the **class hierarchy**, which defines the inheritance relationships between classes. In Java, the class hierarchy begins with class `Object` (in package `java.lang`), which *every* class in Java directly or indirectly **extends** (or "inherits from"). Section G.5 lists the methods of class `Object` that are inherited by all other Java classes.

We distinguish between the ***is-a*** **relationship** and the ***has-a*** **relationship**. *Is-a* represents inheritance. In an *is-a* relationship, *an object of a subclass can also be treated as an object of its superclass*—e.g., a car *is a* vehicle. By contrast, *has-a* represents composition (see Appendix F). In a *has-a* relationship, *an object contains as members references to other objects*—e.g., a car *has a* steering wheel (and a car object has a reference to a steering-wheel object).

Later in the appendix, we discuss the concept of polymorphism, which simplifies programming with objects from the same class hierarchy. You'll see that polymorphism also makes it possible to extend systems to add new capabilities. Finally, we discuss interfaces, which are useful for assigning common functionality to possibly *unrelated* classes. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls in their own customized way.

G.2 Superclasses and Subclasses

Often, an object of one class *is an* object of another class as well. Figure G.1 lists several examples of superclasses and subclasses—superclasses tend to be "more general" and subclasses "more specific." For example, a `CarLoan` *is a* `Loan` as are `HomeImprovementLoans` and `MortgageLoans`. Thus, in Java, class `CarLoan` can be said to inherit from class `Loan`. In this context, class `Loan` is a superclass and class `CarLoan` is a subclass. A `CarLoan` *is a* specific type of `Loan`, but it's incorrect to claim that every `Loan` *is a* `CarLoan`—the `Loan` could be any type of loan.

Superclass	Subclasses
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`, `Sphere`, `Cube`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`BankAccount`	`CheckingAccount`, `SavingsAccount`

Fig. G.1 | Inheritance examples.

Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is often larger than the set of objects represented by any of its subclasses. For example, the superclass `Vehicle` represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, subclass `Car` represents a smaller, more specific subset of vehicles.

University Community Member Hierarchy

Inheritance relationships form treelike hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. Let's develop a sample class hierarchy (Fig. G.2), also called an **inheritance hierarchy**. A university community has thousands of members, including employees, students and alumni. Employees are either faculty or staff members.

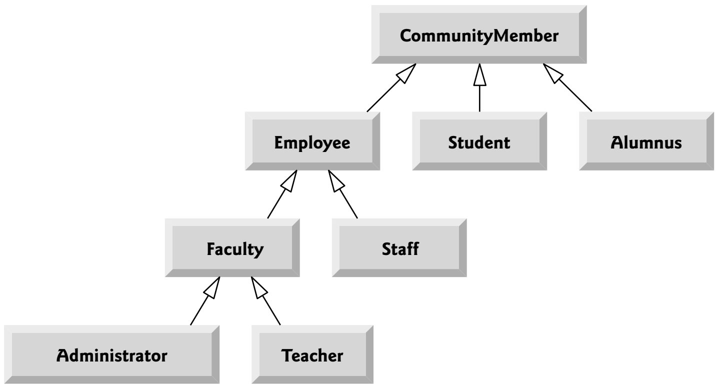


Fig. G.2 | Inheritance hierarchy for university `CommunityMembers`.

Faculty members are either administrators (e.g., deans and department chairpersons) or teachers. The hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors or seniors.

Each arrow in the hierarchy represents an *is-a* relationship. As we follow the arrows upward in this class hierarchy, we can state, for instance, that “an `Employee` *is a* `CommunityMember`” and “a `Teacher` *is a* `Faculty` member.” `CommunityMember` is the direct superclass of `Employee`, `Student` and `Alumnus` and is an indirect superclass of all the other classes in the diagram. Starting from the bottom, you can follow the arrows and apply the *is-a* relationship up to the topmost superclass. For example, an `Administrator` *is a* `Faculty` member, *is an* `Employee`, *is a* `CommunityMember` and, of course, *is an* `Object`.

Shape Hierarchy

Now consider the `Shape` inheritance hierarchy in Fig. G.3. This hierarchy begins with superclass `Shape`, which is extended by subclasses `TwoDimensionalShape` and `ThreeDimensionalShape`—`Shape`s are either `TwoDimensionalShape`s or `ThreeDimensionalShape`s. The third level of this hierarchy contains specific types of `TwoDimensionalShape`s and `ThreeDimensionalShape`s. As in Fig. G.2, we can follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several *is-a* relationships. For instance, a `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`, while a `Sphere` *is a* `ThreeDimensionalShape` and *is a* `Shape`. This hierarchy could contain many other classes. For example, ellipses and trapezoids are `TwoDimensionalShape`s.

It’s possible to treat superclass objects and subclass objects similarly—their commonalities are expressed in the superclass’s members. Objects of all classes that extend a common superclass can be treated as objects of that superclass—such objects have an *is-a* relationship with the superclass. Later in this appendix, we consider many examples that take advantage of the *is-a* relationship.

A subclass can customize methods that it inherits from its superclass. To do this, the subclass **overrides** (redefines) the superclass method with an appropriate implementation, as we’ll see often in this appendix’s code examples.

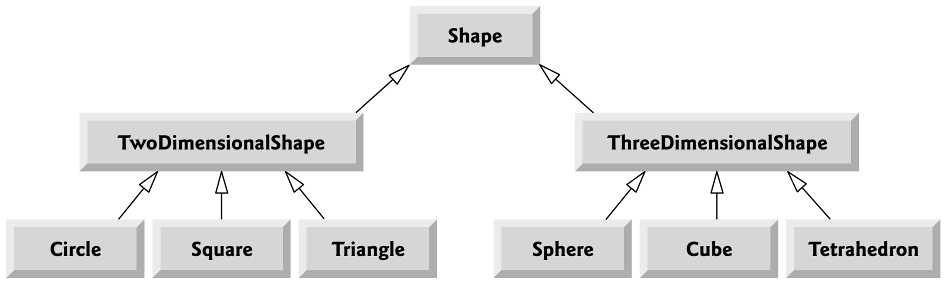


Fig. G.3 | Inheritance hierarchy for `Shape`s.

G.3 `protected` Members

In this section, we introduce access modifier **`protected`**. Using `protected` access offers an intermediate level of access between `public` and `private`. A superclass’s `protected` members can be accessed by the class, by members of its subclasses and by members of other classes in the same package—protected members also have package access.

All public and protected superclass members retain their original access modifier when they become members of the subclass—public members of the superclass become public members of the subclass, and protected members of the superclass become protected members of the subclass. A superclass's private members are *not* accessible outside the class itself. Rather, they're *hidden* in its subclasses and can be accessed only through the public or protected methods inherited from the superclass.

Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names. When a subclass method overrides an inherited superclass method, the *superclass* method can be accessed from the *subclass* by preceding the superclass method name with keyword **super** and a dot (.) separator. We discuss accessing overridden members of the superclass in Section G.4.

G.4 Relationship between Superclasses and Subclasses

We now use an inheritance hierarchy containing types of employees in a company's payroll application to discuss the relationship between a superclass and its subclass. In this company, commission employees (who will be represented as objects of a superclass) are paid a percentage of their sales, while base-salaried commission employees (who will be represented as objects of a subclass) receive a base salary *plus* a percentage of their sales.

We create an example that sets the CommissionEmployee instance variables to private to enforce good software engineering. Then we show how the BasePlusCommissionEmployee subclass can use CommissionEmployee's public methods to manipulate (in a controlled manner) the private instance variables inherited from CommissionEmployee.

G.4.1 Creating and Using a CommissionEmployee Class

We begin by declaring class CommissionEmployee (Fig. G.4). Line 4 begins the class declaration and indicates that class CommissionEmployee **extends** (i.e., inherits from) class **Object** (from package java.lang). This causes class CommissionEmployee to inherit the class Object's methods—class Object does not have any fields. If you don't explicitly specify which class a new class extends, the class extends Object implicitly. For this reason, you typically will not include "extends Object" in your code—we do so in this example only for demonstration purposes.

Overview of Class CommissionEmployee's Methods and Instance Variables

Class CommissionEmployee's public services include a constructor (lines 13–22) and methods earnings (lines 93–96) and toString (lines 99–107). Lines 25–90 declare public *get* and *set* methods for the class's instance variables (declared in lines 6–10) firstName, lastName, socialSecurityNumber, grossSales and commissionRate. The class declares its instance variables as private, so objects of other classes cannot directly access these variables. Declaring instance variables as private and providing *get* and *set* methods to manipulate and validate them helps enforce good software engineering. Methods setGrossSales and setCommissionRate, for example, validate their arguments before assigning the values to instance variables grossSales and commissionRate, respectively. In a real-world, business-critical application, we'd also perform validation in the class's other *set* methods.

```java
// Fig. G.4: CommissionEmployee.java
// CommissionEmployee class represents an employee paid a
// percentage of gross sales.
public class CommissionEmployee extends Object
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last; // should validate
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber
```

Fig. G.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 3.)

```

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
      return commissionRate * grossSales;
   } // end method earnings

   // return String representation of CommissionEmployee object
   @Override // indicates that this method overrides a superclass method
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
         "commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
```

Fig. G.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 3.)

```
105          "gross sales", grossSales,
106          "commission rate", commissionRate );
107    } // end method toString
108 } // end class CommissionEmployee
```

Fig. G.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 3 of 3.)

Class `CommissionEmployee`*'s Constructor*

Constructors are *not* inherited, so class `CommissionEmployee` does not inherit class `Object`'s constructor. However, a superclass's constructors are still available to subclasses. In fact, *the first task of any subclass constructor is to call its direct superclass's constructor*, either explicitly or implicitly (if no constructor call is specified), to ensure that the instance variables inherited from the superclass are initialized properly. In this example, class `CommissionEmployee`'s constructor calls class `Object`'s constructor implicitly. The syntax for calling a superclass constructor explicitly is discussed in Section G.4.3. If the code does not include an explicit call to the superclass constructor, Java *implicitly* calls the superclass's default or no-argument constructor. The comment in line 16 of Fig. G.4 indicates where the implicit call to the superclass `Object`'s default constructor is made (you do not write the code for this call). `Object`'s default (empty) constructor does nothing. Even if a class does not have constructors, the default constructor that the compiler implicitly declares for the class will call the superclass's default or no-argument constructor.

After the implicit call to `Object`'s constructor, lines 17–21 of `CommissionEmployee`'s constructor assign values to the class's instance variables. We do not validate the values of arguments `first`, `last` and `ssn` before assigning them to the corresponding instance variables. We could validate the first and last names—perhaps to ensure that they're of a reasonable length. Similarly, a social security number could be validated using regular expressions to ensure that it contains nine digits, with or without dashes (e.g., `123-45-6789` or `123456789`).

Class `CommissionEmployee`*'s* `earnings` *Method*

Method `earnings` (lines 93–96) calculates a `CommissionEmployee`'s earnings. Line 95 multiplies the `commissionRate` by the `grossSales` and returns the result.

Class `CommissionEmployee`*'s* `toString` *Method and the* `@Override` *Annotation*

Method `toString` (lines 99–107) is special—it's one of the methods that *every* class inherits directly or indirectly from class `Object` (summarized in Section G.5). Method `toString` returns a `String` representing an object. It's called implicitly whenever an object must be converted to a `String` representation, such as when an object is output by `printf` or output by `String` method `format` via the `%s` format specifier. Class `Object`'s `toString` method returns a `String` that includes the name of the object's class. It's primarily a placeholder that can be overridden by a subclass to specify an appropriate `String` representation of the data in a subclass object. Method `toString` of class `CommissionEmployee` overrides (redefines) class `Object`'s `toString` method. When invoked, `CommissionEmployee`'s `toString` method uses `String` method `format` to return a `String` containing information about the `CommissionEmployee`. To override a superclass method, a subclass must declare a method with the same signature (method name, number of parameters, parameter types and order of parameter types) as the superclass method—`Object`'s `toString` method takes no parameters, so `CommissionEmployee` declares `toString` with no parameters.

Line 99 uses the **`@Override` annotation** to indicate that method `toString` should override a superclass method. Annotations have several purposes. For example, when you attempt to override a superclass method, common errors include naming the subclass method incorrectly, or using the wrong number or types of parameters in the parameter list. Each of these problems creates an *unintentional overload* of the superclass method. If you then attempt to call the method on a subclass object, the superclass's version is invoked and the subclass version is ignored—potentially leading to subtle logic errors. When the compiler encounters a method declared with `@Override`, it compares the method's signature with the superclass's method signatures. If there isn't an exact match, the compiler issues an error message, such as "method does not override or implement a method from a supertype." This indicates that you've accidentally overloaded a superclass method. You can then fix your method's signature so that it matches one in the superclass.

In web applications and web services, annotations can also add complex support code to your classes to simplify the development process and can be used by servers to configure certain aspects of web applications.

Common Programming Error G.1

It's a syntax error to override a method with a more restricted access modifier—a `public` method of the superclass cannot become a `protected` or `private` method in the subclass; a `protected` method of the superclass cannot become a `private` method in the subclass. Doing so would break the is-a *relationship in which it's required that all subclass objects be able to respond to method calls that are made to `public` methods declared in the superclass. If a `public` method, for example, could be overridden as a `protected` or `private` method, the subclass objects would not be able to respond to the same method calls as superclass objects. Once a method is declared `public` in a superclass, the method remains `public` for all that class's direct and indirect subclasses.*

Class `CommissionEmployeeTest`

Figure G.5 tests class `CommissionEmployee`. Lines 9–10 instantiate a `CommissionEmployee` object and invoke `CommissionEmployee`'s constructor (lines 13–22 of Fig. G.4) to initialize it with `"Sue"` as the first name, `"Jones"` as the last name, `"222-22-2222"` as the social security number, `10000` as the gross sales amount and `.06` as the commission rate. Lines 15–24 use `CommissionEmployee`'s *get* methods to retrieve the object's instance-variable values for output. Lines 26–27 invoke the object's methods `setGrossSales` and `setCommissionRate` to change the values of instance variables `grossSales` and `commissionRate`. Lines 29–30 output the `String` representation of the updated `CommissionEmployee`. When an object is output using the `%s` format specifier, the object's `toString` method is invoked implicitly to obtain the object's `String` representation. [*Note:* Early in this appendix, we do not use the `earnings` methods of our classes—they're used extensively in the polymorphism part of the appendix.]

```
1  // Fig. G.5: CommissionEmployeeTest.java
2  // CommissionEmployee class test program.
3
```

Fig. G.5 | `CommissionEmployee` class test program. (Part 1 of 2.)

```java
public class CommissionEmployeeTest
{
   public static void main( String[] args )
   {
      // instantiate CommissionEmployee object
      CommissionEmployee employee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // get commission employee data
      System.out.println(
         "Employee information obtained by get methods: \n" );
      System.out.printf( "%s %s\n", "First name is",
         employee.getFirstName() );
      System.out.printf( "%s %s\n", "Last name is",
         employee.getLastName() );
      System.out.printf( "%s %s\n", "Social security number is",
         employee.getSocialSecurityNumber() );
      System.out.printf( "%s %.2f\n", "Gross sales is",
         employee.getGrossSales() );
      System.out.printf( "%s %.2f\n", "Commission rate is",
         employee.getCommissionRate() );

      employee.setGrossSales( 500 ); // set gross sales
      employee.setCommissionRate( .1 ); // set commission rate

      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained by toString", employee );
   } // end main
} // end class CommissionEmployeeTest
```

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Fig. G.5 | `CommissionEmployee` class test program. (Part 2 of 2.)

G.4.2 Creating and Using a `BasePlusCommissionEmployee` Class

We now discuss the second part of our introduction to inheritance by declaring and testing (a completely new and independent) class `BasePlusCommissionEmployee` (Fig. G.6), which contains a first name, last name, social security number, gross sales amount, commission rate *and* base salary. Class `BasePlusCommissionEmployee`'s `public` services include a `BasePlusCommissionEmployee` constructor (lines 15–25) and methods `earnings` (lines 112–115) and `toString` (lines 118–127). Lines 28–109 declare `public` *get* and *set* methods for the class's `private` instance variables (declared in lines 7–12) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` *and* `baseSalary`. These variables and methods encapsulate all the necessary features of a base-salaried commission employee. Note the *similarity* between this class and class `CommissionEmployee` (Fig. G.4)—in this example, we'll not yet exploit that similarity.

```
 1  // Fig. G.6: BasePlusCommissionEmployee.java
 2  // BasePlusCommissionEmployee class represents an employee who receives
 3  // a base salary in addition to a commission.
 4
 5  public class BasePlusCommissionEmployee
 6  {
 7     private String firstName;
 8     private String lastName;
 9     private String socialSecurityNumber;
10     private double grossSales; // gross weekly sales
11     private double commissionRate; // commission percentage
12     private double baseSalary; // base salary per week
13
14     // six-argument constructor
15     public BasePlusCommissionEmployee( String first, String last,
16        String ssn, double sales, double rate, double salary )
17     {
18        // implicit call to Object constructor occurs here
19        firstName = first;
20        lastName = last;
21        socialSecurityNumber = ssn;
22        setGrossSales( sales ); // validate and store gross sales
23        setCommissionRate( rate ); // validate and store commission rate
24        setBaseSalary( salary ); // validate and store base salary
25     } // end six-argument BasePlusCommissionEmployee constructor
26
27     // set first name
28     public void setFirstName( String first )
29     {
30        firstName = first; // should validate
31     } // end method setFirstName
32
33     // return first name
34     public String getFirstName()
35     {
36        return firstName;
37     } // end method getFirstName
38
39     // set last name
40     public void setLastName( String last )
41     {
42        lastName = last; // should validate
43     } // end method setLastName
```

Fig. G.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 3.)

```java

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate
```

Fig. G.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 3.)

```
 94
 95      // set base salary
 96      public void setBaseSalary( double salary )
 97      {
 98         if ( salary >= 0.0 )
 99            baseSalary = salary;
100         else
101            throw new IllegalArgumentException(
102               "Base salary must be >= 0.0" );
103      } // end method setBaseSalary
104
105      // return base salary
106      public double getBaseSalary()
107      {
108         return baseSalary;
109      } // end method getBaseSalary
110
111      // calculate earnings
112      public double earnings()
113      {
114         return baseSalary + ( commissionRate * grossSales );
115      } // end method earnings
116
117      // return String representation of BasePlusCommissionEmployee
118      @Override // indicates that this method overrides a superclass method
119      public String toString()
120      {
121         return String.format(
122            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
123            "base-salaried commission employee", firstName, lastName,
124            "social security number", socialSecurityNumber,
125            "gross sales", grossSales, "commission rate", commissionRate,
126            "base salary", baseSalary );
127      } // end method toString
128   } // end class BasePlusCommissionEmployee
```

Fig. G.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 3.)

Class `BasePlusCommissionEmployee` does not specify "`extends Object`" in line 5, so the class implicitly extends `Object`. Also, like class `CommissionEmployee`'s constructor (lines 13–22 of Fig. G.4), class `BasePlusCommissionEmployee`'s constructor invokes class `Object`'s default constructor implicitly, as noted in the comment in line 18.

Class `BasePlusCommissionEmployee`'s `earnings` method (lines 112–115) returns the result of adding the `BasePlusCommissionEmployee`'s base salary to the product of the commission rate and the employee's gross sales.

Class `BasePlusCommissionEmployee` overrides `Object` method `toString` to return a `String` containing the `BasePlusCommissionEmployee`'s information. Once again, we use format specifier `%.2f` to format the gross sales, commission rate and base salary with two digits of precision to the right of the decimal point (line 122).

Testing Class **`BasePlusCommissionEmployee`**

Figure G.7 tests class `BasePlusCommissionEmployee`. Lines 9–11 create a `BasePlusCommissionEmployee` object and pass `"Bob"`, `"Lewis"`, `"333-33-3333"`, `5000`, `.04` and `300` to the constructor as the first name, last name, social security number, gross sales, commission rate and base salary, respectively. Lines 16–27 use `BasePlusCommissionEmployee`'s *get* methods to retrieve the values of the object's instance variables for output. Line 29 invokes the object's `setBaseSalary` method to change the base salary. Method `setBaseSalary` (Fig. G.6, lines 96–103) ensures that instance variable `baseSalary` is not assigned a negative value. Lines 31–33 of Fig. G.7 invoke method `toString` explicitly to get the object's `String` representation.

Notes on Class **`BasePlusCommissionEmployee`**

Much of class `BasePlusCommissionEmployee`'s code (Fig. G.6) is similar, or identical, to that of class `CommissionEmployee` (Fig. G.4). For example, `private` instance variables

```java
 1  // Fig. G.7: BasePlusCommissionEmployeeTest.java
 2  // BasePlusCommissionEmployee test program.
 3
 4  public class BasePlusCommissionEmployeeTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // instantiate BasePlusCommissionEmployee object
 9        BasePlusCommissionEmployee employee =
10           new BasePlusCommissionEmployee(
11           "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13        // get base-salaried commission employee data
14        System.out.println(
15           "Employee information obtained by get methods: \n" );
16        System.out.printf( "%s %s\n", "First name is",
17           employee.getFirstName() );
18        System.out.printf( "%s %s\n", "Last name is",
19           employee.getLastName() );
20        System.out.printf( "%s %s\n", "Social security number is",
21           employee.getSocialSecurityNumber() );
22        System.out.printf( "%s %.2f\n", "Gross sales is",
23           employee.getGrossSales() );
24        System.out.printf( "%s %.2f\n", "Commission rate is",
25           employee.getCommissionRate() );
26        System.out.printf( "%s %.2f\n", "Base salary is",
27           employee.getBaseSalary() );
28
29        employee.setBaseSalary( 1000 ); // set base salary
30
31        System.out.printf( "\n%s:\n\n%s\n",
32           "Updated employee information obtained by toString",
33            employee.toString() );
34     } // end main
35  } // end class BasePlusCommissionEmployeeTest
```

Fig. G.7 | `BasePlusCommissionEmployee` test program. (Part 1 of 2.)

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Fig. G.7 | `BasePlusCommissionEmployee` test program. (Part 2 of 2.)

`firstName` and `lastName` and methods `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical to those of class `CommissionEmployee`. The classes also both contain `private` instance variables `socialSecurityNumber`, `commissionRate` and `grossSales`, and corresponding *get* and *set* methods. In addition, the `BasePlusCommissionEmployee` constructor is almost identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`. The other additions to class `BasePlusCommissionEmployee` are `private` instance variable `baseSalary` and methods `setBaseSalary` and `getBaseSalary`. Class `BasePlusCommissionEmployee`'s `toString` method is nearly identical to that of class `CommissionEmployee` except that it also outputs instance variable `baseSalary` with two digits of precision to the right of the decimal point.

We literally *copied* code from class `CommissionEmployee` and *pasted* it into class `BasePlusCommissionEmployee`, then modified class `BasePlusCommissionEmployee` to include a base salary and methods that manipulate the base salary. This *"copy-and-paste" approach* is often error prone and time consuming. Worse yet, it spreads copies of the same code throughout a system, creating a code-maintenance nightmare. Is there a way to "absorb" the instance variables and methods of one class in a way that makes them part of other classes *without duplicating code*? Next we answer this question, using a more elegant approach to building classes that emphasizes the benefits of inheritance.

Software Engineering Observation G.1

With inheritance, the common *instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are made for these common features in the superclass—subclasses then inherit the changes. Without inheritance, changes would need to be made to* all *the source-code files that contain a* copy *of the code in question.*

G.4.3 Creating a `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy

Now we redeclare class `BasePlusCommissionEmployee` (Fig. G.8) to *extend* class `CommissionEmployee` (Fig. G.4). A `BasePlusCommissionEmployee` object *is a* `CommissionEmployee`, because inheritance passes on class `CommissionEmployee`'s capabilities. Class `BasePlus-CommissionEmployee` also has instance variable `baseSalary` (Fig. G.8, line 6).

Keyword extends (line 4) indicates inheritance. BasePlusCommissionEmployee *inherits* CommissionEmployee's instance variables and methods, but only the superclass's public and protected members are directly accessible in the subclass. The CommissionEmployee constructor is *not* inherited. So, the public BasePlusCommissionEmployee services include its constructor (lines 9–16), public methods inherited from CommissionEmployee, and methods setBaseSalary (lines 19–26), getBaseSalary (lines 29–32), earnings (lines 35–40) and toString (lines 43–53). Methods earnings and toString *override* the corresponding methods in class CommissionEmployee because their superclass versions do not properly calculate a BasePlusCommissionEmployee's earnings or return an appropriate String representation.

```java
 1  // Fig. G.8: BasePlusCommissionEmployee.java
 2  // private superclass members cannot be accessed in a subclass.
 3
 4  public class BasePlusCommissionEmployee extends CommissionEmployee
 5  {
 6     private double baseSalary; // base salary per week
 7
 8     // six-argument constructor
 9     public BasePlusCommissionEmployee( String first, String last,
10        String ssn, double sales, double rate, double salary )
11     {
12        // explicit call to superclass CommissionEmployee constructor
13        super( first, last, ssn, sales, rate );
14
15        setBaseSalary( salary ); // validate and store base salary
16     } // end six-argument BasePlusCommissionEmployee constructor
17
18     // set base salary
19     public void setBaseSalary( double salary )
20     {
21        if ( salary >= 0.0 )
22           baseSalary = salary;
23        else
24           throw new IllegalArgumentException(
25              "Base salary must be >= 0.0" );
26     } // end method setBaseSalary
27
28     // return base salary
29     public double getBaseSalary()
30     {
31        return baseSalary;
32     } // end method getBaseSalary
33
34     // calculate earnings
35     @Override // indicates that this method overrides a superclass method
36     public double earnings()
37     {
38        // not allowed: commissionRate and grossSales private in superclass
39        return baseSalary + ( commissionRate * grossSales );
40     } // end method earnings
```

Fig. G.8 | private superclass members cannot be accessed in a subclass. (Part 1 of 2.)

```
41
42     // return String representation of BasePlusCommissionEmployee
43     @Override // indicates that this method overrides a superclass method
44     public String toString()
45     {
46        // not allowed: attempts to access private superclass members
47        return String.format(
48           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
49           "base-salaried commission employee", firstName, lastName,
50           "social security number", socialSecurityNumber,
51           "gross sales", grossSales, "commission rate", commissionRate,
52           "base salary", baseSalary );
53     } // end method toString
54  } // end class BasePlusCommissionEmployee
```

```
BasePlusCommissionEmployee.java:39: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee.java:39: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee.java:49: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee.java:49: lastName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee.java:50: socialSecurityNumber has private access
in CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:51: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee.java:51: commissionRate has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                       ^
7 errors
```

Fig. G.8 | `private` superclass members cannot be accessed in a subclass. (Part 2 of 2.)

A Subclass's Constructor Must Call Its Superclass's Constructor

Each subclass constructor must implicitly or explicitly call its superclass constructor to initialize the instance variables inherited from the superclass. Line 13 in `BasePlusCommissionEmployee`'s six-argument constructor (lines 9–16) explicitly calls class `CommissionEmployee`'s five-argument constructor (declared at lines 13–22 of Fig. G.4) to initialize the superclass portion of a `BasePlusCommissionEmployee` object (i.e., variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`). We do this by using the **superclass constructor call syntax**—keyword `super`, followed by a set of parentheses containing the superclass constructor arguments. The arguments `first`, `last`, `ssn`, `sales` and `rate` are used to initialize superclass members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`, respectively. If `BasePlusCommissionEmployee`'s constructor did not invoke the superclass's constructor explicitly, Java would attempt to invoke the superclass's no-argument or default constructor. Class `CommissionEmployee` does not have such a constructor, so the compiler would issue an error. The explicit superclass constructor call in line 13 of Fig. G.8 must be the *first* statement in the subclass constructor's body. When a superclass contains a no-argument constructor, you can use `super()` to call that constructor explicitly, but this is rarely done.

`BasePlusCommissionEmployee` Method `Earnings`

The compiler generates errors for line 39 because superclass `CommissionEmployee`'s instance variables `commissionRate` and `grossSales` are `private`—subclass `BasePlusCommissionEmployee`'s methods are not allowed to access superclass `CommissionEmployee`'s `private` instance variables. We highlighted the erroneous code. The compiler issues additional errors at lines 49–51 of `BasePlusCommissionEmployee`'s `toString` method for the same reason. The errors in `BasePlusCommissionEmployee` could have been prevented by using the *get* methods inherited from class `CommissionEmployee`. For example, line 39 could have used `getCommissionRate` and `getGrossSales` to access `CommissionEmployee`'s `private` instance variables `commissionRate` and `grossSales`, respectively. Lines 49–51 also could have used appropriate *get* methods to retrieve the values of the superclass's instance variables.

G.4.4 `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy Using `protected` Instance Variables

To enable class `BasePlusCommissionEmployee` to directly access superclass instance variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`, we can declare those members as `protected` in the superclass. As we discussed in Section G.3, a superclass's `protected` members are accessible by all subclasses of that superclass. In the new `CommissionEmployee` class, we modified only lines 6–10 of Fig. G.4 to declare the instance variables with the `protected` access modifier as follows:

```
    protected String firstName;
    protected String lastName;
    protected String socialSecurityNumber;
    protected double grossSales; // gross weekly sales
    protected double commissionRate; // commission percentage
```

The rest of the class declaration (which is not shown here) is identical to that of Fig. G.4.

We could have declared `CommissionEmployee`'s instance variables `public` to enable subclass `BasePlusCommissionEmployee` to access them. However, declaring `public` instance variables is poor software engineering because it allows unrestricted access to the these variables, greatly increasing the chance of errors. With `protected` instance variables, the subclass gets access to the instance variables, but classes that are not subclasses and classes that are not in the same package cannot access these variables directly—recall that `protected` class members are also visible to other classes in the same package.

Class **BasePlusCommissionEmployee**

Class BasePlusCommissionEmployee (Fig. G.9) extends the new version of class CommissionEmployee with protected instance variables. BasePlusCommissionEmployee objects inherit CommissionEmployee's protected instance variables firstName, lastName, socialSecurityNumber, grossSales and commissionRate—all these variables are now protected members of BasePlusCommissionEmployee. As a result, the compiler does not generate errors when compiling line 37 of method earnings and lines 46–48 of method toString. If another class extends this version of class BasePlusCommissionEmployee, the new subclass also can access the protected members.

```java
 1  // Fig. G.9: BasePlusCommissionEmployee.java
 2  // BasePlusCommissionEmployee inherits protected instance
 3  // variables from CommissionEmployee.
 4
 5  public class BasePlusCommissionEmployee extends CommissionEmployee
 6  {
 7     private double baseSalary; // base salary per week
 8
 9     // six-argument constructor
10     public BasePlusCommissionEmployee( String first, String last,
11        String ssn, double sales, double rate, double salary )
12     {
13        super( first, last, ssn, sales, rate );
14        setBaseSalary( salary ); // validate and store base salary
15     } // end six-argument BasePlusCommissionEmployee constructor
16
17     // set base salary
18     public void setBaseSalary( double salary )
19     {
20        if ( salary >= 0.0 )
21           baseSalary = salary;
22        else
23           throw new IllegalArgumentException(
24              "Base salary must be >= 0.0" );
25     } // end method setBaseSalary
26
27     // return base salary
28     public double getBaseSalary()
29     {
30        return baseSalary;
31     } // end method getBaseSalary
32
33     // calculate earnings
34     @Override // indicates that this method overrides a superclass method
35     public double earnings()
36     {
37        return baseSalary + ( commissionRate * grossSales );
38     } // end method earnings
39
```

Fig. G.9 | BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee. (Part 1 of 2.)

```
40     // return String representation of BasePlusCommissionEmployee
41     @Override // indicates that this method overrides a superclass method
42     public String toString()
43     {
44        return String.format(
45           "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
46           "base-salaried commission employee", firstName, lastName,
47           "social security number", socialSecurityNumber,
48           "gross sales", grossSales, "commission rate", commissionRate,
49           "base salary", baseSalary );
50     } // end method toString
51  } // end class BasePlusCommissionEmployee
```

Fig. G.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 2 of 2.)

When you create a `BasePlusCommissionEmployee` object, it contains all instance variables declared in the class hierarchy to that point—i.e., those from classes `Object`, `CommissionEmployee` and `BasePlusCommissionEmployee`. Class `BasePlusCommissionEmployee` does not inherit class `CommissionEmployee`'s constructor. However, class `BasePlusCommissionEmployee`'s six-argument constructor (lines 10–15) calls class `CommissionEmployee`'s five-argument constructor *explicitly* to initialize the instance variables that `BasePlusCommissionEmployee` inherited from class `CommissionEmployee`. Similarly, class `CommissionEmployee`'s constructor *implicitly* calls class `Object`'s constructor. `BasePlusCommissionEmployee`'s constructor must do this *explicitly* because `CommissionEmployee` does *not* provide a no-argument constructor that could be invoked implicitly.

Testing Class `BasePlusCommissionEmployee`

The `BasePlusCommissionEmployeeTest` class for this example is identical to that of Fig. G.7 and produces the same output, so we do not show it here. Although the version of class `BasePlusCommissionEmployee` in Fig. G.6 does not use inheritance and the version in Fig. G.9 does, *both classes provide the same functionality*. The source code in Fig. G.9 (51 lines) is considerably shorter than that in Fig. G.6 (128 lines), because most of `BasePlusCommissionEmployee`'s functionality is now inherited from `CommissionEmployee`—there's now only one copy of the `CommissionEmployee` functionality. This makes the code easier to maintain, modify and debug, because the code related to a commission employee exists only in class `CommissionEmployee`.

Notes on Using `protected` *Instance Variables*

In this example, we declared superclass instance variables as `protected` so that subclasses could access them. Inheriting `protected` instance variables slightly increases performance, because we can directly access the variables in the subclass without incurring the overhead of a *set* or *get* method call. In most cases, however, it's better to use `private` instance variables to encourage proper software engineering, and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.

Using `protected` instance variables creates several potential problems. First, the subclass object can set an inherited variable's value directly without using a *set* method. Therefore, a subclass object can assign an invalid value to the variable, possibly leaving the object in an inconsistent state. For example, if we were to declare `CommissionEmployee`'s instance variable `grossSales` as `protected`, a subclass object (e.g., `BasePlusCommissionEmployee`) could then assign a negative value to `grossSales`. Another problem with using `protected` instance variables is that subclass methods are more likely to be written so that they depend on the superclass's data implementation. In practice, subclasses should depend only on the superclass services (i.e., non-`private` methods) and not on the superclass data implementation. With `protected` instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes. For example, if for some reason we were to change the names of instance variables `firstName` and `lastName` to `first` and `last`, then we would have to do so for all occurrences in which a subclass directly references superclass instance variables `firstName` and `lastName`. In such a case, the software is said to be **fragile** or **brittle**, because a small change in the superclass can "break" subclass implementation. You should be able to change the superclass implementation while still providing the same services to the subclasses. Of course, if the superclass services change, we must reimplement our subclasses. A third problem is that a class's `protected` members are visible to all classes in the same package as the class containing the `protected` members—this is not always desirable.

Software Engineering Observation G.2

Use the `protected` access modifier when a superclass should provide a method only to its subclasses and other classes in the same package, but not to other clients.

Software Engineering Observation G.3

Declaring superclass instance variables `private` (as opposed to `protected`) enables the superclass implementation of these instance variables to change without affecting subclass implementations.

Error-Prevention Tip G.1

When possible, do not include `protected` instance variables in a superclass. Instead, include non-`private` methods that access `private` instance variables. This will help ensure that objects of the class maintain consistent states.

G.4.5 `CommissionEmployee`–`BasePlusCommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables

Let's reexamine our hierarchy once more, this time using good software engineering practices. Class `CommissionEmployee` (Fig. G.10) declares instance variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as *`private`* (lines 6–10) and provides `public` methods `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `toString` for manipulating these values. Methods `earnings` (lines 93–96) and `toString` (lines 99–107) use the class's *get* methods to obtain the values of its instance variables. If we decide to change the instance-variable names, the `earnings` and `toString` declarations will not require modification—only the bodies of the *get* and *set* methods that directly manipulate the instance variables will need to change. These changes occur solely within the superclass—no changes to the subclass are needed. *Localizing the effects of changes* like this is a good software engineering practice.

```java
// Fig. G.10: CommissionEmployee.java
// CommissionEmployee class uses methods to manipulate its
// private instance variables.
public class CommissionEmployee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      la5stName = last; // should validate
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber
```

Fig. G.10 | CommissionEmployee class uses methods to manipulate its private instance variables. (Part 1 of 3.)

```

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

   // return String representation of CommissionEmployee object
   @Override // indicates that this method overrides a superclass method
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
         "commission employee", getFirstName(), getLastName(),
         "social security number", getSocialSecurityNumber(),
```

Fig. G.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 2 of 3.)

```
105           "gross sales", getGrossSales(),
106           "commission rate", getCommissionRate() );
107    } // end method toString
108 } // end class CommissionEmployee
```

Fig. G.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 3 of 3.)

Subclass `BasePlusCommissionEmployee` (Fig. G.11) inherits `CommissionEmployee`'s non-`private` methods and can access the `private` superclass members via those methods. Class `BasePlusCommissionEmployee` has several changes that distinguish it from Fig. G.9. Methods `earnings` (lines 35–39) and `toString` (lines 42–47) each invoke method `getBaseSalary` to obtain the base salary value, rather than accessing `baseSalary` directly. If we decide to rename instance variable `baseSalary`, only the bodies of method `setBaseSalary` and `getBaseSalary` will need to change.

```
 1  // Fig. G.11: BasePlusCommissionEmployee.java
 2  // BasePlusCommissionEmployee class inherits from CommissionEmployee
 3  // and accesses the superclass's private data via inherited
 4  // public methods.
 5
 6  public class BasePlusCommissionEmployee extends CommissionEmployee
 7  {
 8     private double baseSalary; // base salary per week
 9
10     // six-argument constructor
11     public BasePlusCommissionEmployee( String first, String last,
12        String ssn, double sales, double rate, double salary )
13     {
14        super( first, last, ssn, sales, rate );
15        setBaseSalary( salary ); // validate and store base salary
16     } // end six-argument BasePlusCommissionEmployee constructor
17
18     // set base salary
19     public void setBaseSalary( double salary )
20     {
21        if ( salary >= 0.0 )
22           baseSalary = salary;
23        else
24           throw new IllegalArgumentException(
25              "Base salary must be >= 0.0" );
26     } // end method setBaseSalary
27
28     // return base salary
29     public double getBaseSalary()
30     {
31        return baseSalary;
32     } // end method getBaseSalary
33
```

Fig. G.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 1 of 2.)

```
34      // calculate earnings
35      @Override // indicates that this method overrides a superclass method
36      public double earnings()
37      {
38         return getBaseSalary() + super.earnings();
39      } // end method earnings
40
41      // return String representation of BasePlusCommissionEmployee
42      @Override // indicates that this method overrides a superclass method
43      public String toString()
44      {
45         return String.format( "%s %s\n%s: %.2f", "base-salaried",
46            super.toString(), "base salary", getBaseSalary() );
47      } // end method toString
48   } // end class BasePlusCommissionEmployee
```

Fig. G.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 2 of 2.)

Class **`BasePlusCommissionEmployee`***'s* **`earnings`** *Method*

Method `earnings` (lines 35–39) overrides class `CommissionEmployee`'s `earnings` method (Fig. G.10, lines 93–96) to calculate a base-salaried commission employee's earnings. The new version obtains the portion of the earnings based on commission alone by calling `CommissionEmployee`'s `earnings` method with `super.earnings()` (line 38), then adds the base salary to this value to calculate the total earnings. Note the syntax used to invoke an overridden superclass method from a subclass—place the keyword `super` and a dot (`.`) separator before the superclass method name. This method invocation is a good software engineering practice—if a method performs all or some of the actions needed by another method, call that method rather than duplicate its code. By having `BasePlusCommissionEmployee`'s `earnings` method invoke `CommissionEmployee`'s `earnings` method to calculate part of a `BasePlusCommissionEmployee` object's earnings, we *avoid duplicating the code* and *reduce code-maintenance problems*. If we did not use "`super.`" then `BasePlusCommissionEmployee`'s `earnings` method would *call itself* rather than the superclass version. This would result in a phenomenon called *infinite recursion*, which would eventually cause the method-call stack to overflow—a fatal runtime error.

Class **`BasePlusCommissionEmployee`***'s* **`toString`** *Method*

Similarly, `BasePlusCommissionEmployee`'s `toString` method (Fig. G.11, lines 42–47) overrides class `CommissionEmployee`'s `toString` method (Fig. G.10, lines 99–107) to return a `String` representation that's appropriate for a base-salaried commission employee. The new version creates part of a `BasePlusCommissionEmployee` object's `String` representation (i.e., the `String` `"commission employee"` and the values of class `CommissionEmployee`'s `private` instance variables) by calling `CommissionEmployee`'s `toString` method with the expression `super.toString()` (Fig. G.11, line 46). `BasePlusCommissionEmployee`'s `toString` method then outputs the remainder of a `BasePlusCommissionEmployee` object's `String` representation (i.e., the value of class `BasePlusCommissionEmployee`'s base salary).

Common Programming Error G.2

When a superclass method is overridden in a subclass, the subclass version often calls the superclass version to do a portion of the work. Failure to prefix the superclass method name with the keyword `super` and a dot (.) separator when calling the superclass's method causes the subclass method to call itself, potentially creating an error called infinite recursion. Recursion, used correctly, is a powerful capability.

Testing Class `BasePlusCommissionEmployee`

Class `BasePlusCommissionEmployeeTest` performs the same manipulations on a `BasePlusCommissionEmployee` object as in Fig. G.7 and produces the same output, so we do not show it here. Although each `BasePlusCommissionEmployee` class you've seen behaves identically, the version in Fig. G.11 is the best engineered. By using inheritance and by calling methods that hide the data and ensure consistency, we've efficiently and effectively constructed a well-engineered class.

G.5 Class `Object`

As we discussed earlier in this appendix, all classes in Java inherit directly or indirectly from the `Object` class (package `java.lang`), so its 11 methods (some are overloaded) are inherited by all other classes. Figure G.12 summarizes `Object`'s methods. We discuss several `Object` methods throughout this book (as indicated in Fig. G.12).

Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation performs a so-called **shallow copy**—instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a **deep copy** that creates a new object for each reference-type instance variable. Implementing `clone` correctly is difficult. For this reason, its use is discouraged. Many industry experts suggest that object serialization should be used instead. We introduce object serialization in Appendix J.
`equals`	This method compares two objects for equality and returns `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the *contents* of the two objects. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory.
`finalize`	This `protected` method (introduced in Section F.9) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall that it's unclear whether, or when, method `finalize` will be called. For this reason, most programmers should avoid method `finalize`.

Fig. G.12 | `Object` methods. (Part 1 of 2.)

Method	Description
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`).
`hashCode`	Hashcodes are `int` values that are useful for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (discussed in Section J.9). This method is also called as part of class `Object`'s default `toString` method implementation.
`wait`, `notify`, `notifyAll`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Appendix J.
`toString`	This method (introduced in Section G.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Fig. G.12 | `Object` methods. (Part 2 of 2.)

Recall from Appendix E that arrays are objects. As a result, like all other objects, arrays inherit the members of class `Object`. Every array has an overridden `clone` method that copies the array. However, if the array stores references to objects, the objects are not copied—a *shallow copy is* performed.

G.6 Introduction to Polymorphism

We continue our study of object-oriented programming by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables you to "program in the general" rather than "program in the specific." In particular, polymorphism enables you to write programs that process objects that share the same superclass (either directly or indirectly) as if they're all objects of the superclass; this can simplify programming.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the types of animals under investigation. Imagine that each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x-y* coordinates. Each subclass implements method `move`. Our program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the *same* message once per second—namely, `move`. Each specific type of `Animal` responds to a `move` message in its own way—a `Fish` might swim three feet, a `Frog` might jump five feet and a `Bird` might fly ten feet. Each object knows how to modify its *x-y* coordinates appropriately for its *specific* type of movement. Relying on each object to know how to "do the right thing" (i.e., do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism. The same message (in this case, `move`) sent to a variety of objects has "many forms" of results—hence the term polymorphism.

Programming in the Specific
Occasionally, when performing polymorphic processing, we need to program "in the specific." We'll demonstrate that a program can determine the type of an object at *execution time* and act on that object accordingly.

Interfaces
The appendix continues with an introduction to Java interfaces. An interface describes a set of methods that can be called on an object, but does *not* provide concrete implementations for all the methods. You can declare classes that **implement** (i.e., provide concrete implementations for the methods of) one or more interfaces. Each interface method must be declared in all the classes that explicitly implement the interface. Once a class implements an interface, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface. This is true of all subclasses of that class as well.

Interfaces are particularly useful for assigning common functionality to possibly *unrelated* classes. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls. To demonstrate creating and using interfaces, we modify our payroll application to create a general accounts payable application that can calculate payments due for company employees and invoice amounts to be billed for purchased goods. As you'll see, interfaces enable polymorphic capabilities similar to those possible with inheritance.

G.7 Polymorphism: An Example

Space Objects in a Video Game
Suppose we design a video game that manipulates objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Imagine that each class inherits from the superclass `SpaceObject`, which contains method `draw`. Each subclass implements this method. A screen manager maintains a collection (e.g., a `SpaceObject` array) of references to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same message—namely, `draw`. However, each object responds its own way, based on its class. For example, a `Martian` object might draw itself in red with green eyes and the appropriate number of antennae. A `SpaceShip` object might draw itself as a bright silver flying saucer. A `LaserBeam` object might draw itself as a bright red beam across the screen. Again, the *same* message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code. Suppose that we want to add `Mercurian` objects to our video game. To do so, we'd build a class `Mercurian` that extends `SpaceObject` and provides its own `draw` method implementation. When `Mercurian` objects appear in the `SpaceObject` collection, the screen manager code *invokes method `draw`, exactly as it does for every other object in the collection, regardless of its type.* So the new `Mercurian` objects simply "plug right in" without any modification of the screen manager code by the programmer. Thus, without modifying the system (other than to build new classes and modify the code that creates new objects), you can use polymorphism to conveniently include additional types that were not envisioned when the system was created.

Software Engineering Observation G.4

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics. You can command objects to behave in manners appropriate to those objects, without knowing their types (as long as the objects belong to the same inheritance hierarchy).

Software Engineering Observation G.5

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

G.8 Demonstrating Polymorphic Behavior

Section G.4 created a class hierarchy, in which class `BasePlusCommissionEmployee` inherited from `CommissionEmployee`. The examples in that section manipulated `CommissionEmployee` and `BasePlusCommissionEmployee` objects by using references to them to invoke their methods—we aimed superclass variables at superclass objects and subclass variables at subclass objects. These assignments are natural and straightforward—superclass variables are *intended* to refer to superclass objects, and subclass variables are *intended* to refer to subclass objects. However, as you'll soon see, other assignments are possible.

In the next example, we aim a *superclass* reference at *a subclass* object. We then show how invoking a method on a subclass object via a superclass reference invokes the *subclass* functionality—the type of the *referenced object*, not the type of the *variable*, determines which method is called. This example demonstrates that *an object of a subclass can be treated as an object of its superclass,* enabling various interesting manipulations. A program can create an array of superclass variables that refer to objects of many subclass types. This is allowed because each subclass object *is an* object of its superclass. For instance, we can assign the reference of a `BasePlusCommissionEmployee` object to a superclass `CommissionEmployee` variable, because a `BasePlusCommissionEmployee` *is a* `CommissionEmployee`—we can treat a `BasePlusCommissionEmployee` as a `CommissionEmployee`.

As you'll learn later in this appendix, you *cannot treat a superclass object as a subclass object,* because a superclass object is *not* an object of any of its subclasses. For example, we cannot assign the reference of a `CommissionEmployee` object to a subclass `BasePlusCommissionEmployee` variable, because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`—a `CommissionEmployee` does *not* have a `baseSalary` instance variable and does *not* have methods `setBaseSalary` and `getBaseSalary`. The *is-a* relationship applies only *up the hierarchy* from a subclass to its direct and *indirect* superclasses, and *not* vice versa (i.e., *not down the hierarchy* from a superclass to its subclasses).

The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if we explicitly *cast* the superclass reference to the subclass type—a technique we discuss in Section G.10. Why would we ever want to perform such an assignment? A superclass reference can be used to invoke only the methods declared in the superclass—attempting to invoke subclass-only methods through a superclass reference results in compilation errors. If a program needs to perform a subclass-specific operation on a subclass object referenced by a superclass variable, the program must first cast the superclass reference to a subclass reference through a technique known as **downcasting**. This enables the program to invoke subclass methods that are *not* in the superclass. We show a downcasting example in Section G.10.

The example in Fig. G.13 demonstrates three ways to use superclass and subclass variables to store references to superclass and subclass objects. The first two are straightforward—as in Section G.4, we assign a superclass reference to a superclass variable, and a subclass reference to a subclass variable. Then we demonstrate the relationship between subclasses and superclasses (i.e., the *is-a* relationship) by assigning a subclass reference to a superclass variable. This program uses classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Fig. G.10 and Fig. G.11, respectively.

```java
// Fig. G.13: PolymorphismTest.java
// Assigning superclass and subclass references to superclass and
// subclass variables.

public class PolymorphismTest
{
   public static void main( String[] args )
   {
      // assign superclass reference to superclass variable
      CommissionEmployee commissionEmployee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // assign subclass reference to subclass variable
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call CommissionEmployee's toString with superclass reference ",
         "to superclass object", commissionEmployee.toString() );

      // invoke toString on subclass object using subclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call BasePlusCommissionEmployee's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString() );

      // invoke toString on subclass object using superclass variable
      CommissionEmployee commissionEmployee2 =
         basePlusCommissionEmployee;
      System.out.printf( "%s %s:\n\n%s\n",
         "Call BasePlusCommissionEmployee's toString with superclass",
         "reference to subclass object", commissionEmployee2.toString() );
   } // end main
} // end class PolymorphismTest
```

Fig. G.13 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 2.)

```
Call CommissionEmployee's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. G.13 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 2.)

In Fig. G.13, lines 10–11 create a `CommissionEmployee` object and assign its reference to a `CommissionEmployee` variable. Lines 14–16 create a `BasePlusCommissionEmployee` object and assign its reference to a `BasePlusCommissionEmployee` variable. These assignments are natural—for example, a `CommissionEmployee` variable's primary purpose is to hold a reference to a `CommissionEmployee` object. Lines 19–21 use `commissionEmployee` to invoke `toString` explicitly. Because `commissionEmployee` refers to a `CommissionEmployee` object, superclass `CommissionEmployee`'s version of `toString` is called. Similarly, lines 24–27 use `basePlusCommissionEmployee` to invoke `toString` explicitly on the `BasePlusCommissionEmployee` object. This invokes subclass `BasePlusCommissionEmployee`'s version of `toString`.

Lines 30–31 then assign the reference of subclass object `basePlusCommissionEmployee` to a superclass `CommissionEmployee` variable, which lines 32–34 use to invoke method `toString`. *When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.* Hence, `commissionEmployee2.toString()` in line 34 actually calls class `BasePlusCommissionEmployee`'s `toString` method. The Java compiler allows this "crossover" because an object of a subclass *is an* object of its superclass (but not vice versa). When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the call is compiled. At execution time, the type of the object to which the variable refers determines the actual method to use. This process, called *dynamic binding*, is discussed in detail in Section G.10.

G.9 Abstract Classes and Methods

When we think of a class, we assume that programs will create objects of that type. Sometimes it's useful to declare classes—called **abstract classes**—for which you *never* intend to create objects. Because they're used only as superclasses in inheritance hierarchies, we refer to them as **abstract superclasses**. These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are *incomplete*. Subclasses must declare the "missing pieces" to become "concrete" classes, from which you *can* instantiate objects. Otherwise, these subclasses, too, will be abstract. We demonstrate abstract classes in Section G.10.

Purpose of Abstract Classes

An abstract class's purpose is to provide an appropriate superclass from which other classes can inherit and thus share a common design. In the `Shape` hierarchy of Fig. G.3, for example, subclasses inherit the notion of what it means to be a `Shape`—perhaps common attributes such as `location`, `color` and `borderThickness`, and behaviors such as `draw`, `move`, `resize` and `changeColor`. Classes that can be used to instantiate objects are called **concrete classes**. Such classes provide implementations of *every* method they declare (some of the implementations can be inherited). For example, we could derive concrete classes `Circle`, `Square` and `Triangle` from abstract superclass `TwoDimensionalShape`. Similarly, we could derive concrete classes `Sphere`, `Cube` and `Tetrahedron` from abstract superclass `ThreeDimensionalShape`. Abstract superclasses are *too general* to create real objects—they specify only what is *common* among subclasses. We need to be more *specific* before we can create objects. For example, if you send the `draw` message to abstract class `TwoDimensionalShape`, the class knows that two-dimensional shapes should be *drawable*, but it does not know what *specific* shape to draw, so it cannot implement a real `draw` method. Concrete classes provide the *specifics* that make it reasonable to instantiate objects.

Not all hierarchies contain abstract classes. However, you'll often write client code that uses only abstract superclass types to reduce the client code's dependencies on a range of subclass types. For example, you can write a method with a parameter of an abstract superclass type. When called, such a method can receive an object of *any* concrete class that directly or indirectly extends the superclass specified as the parameter's type.

Abstract classes sometimes constitute several levels of a hierarchy. For example, the `Shape` hierarchy of Fig. G.3 begins with abstract class `Shape`. On the next level of the hierarchy are *abstract* classes `TwoDimensionalShape` and `ThreeDimensionalShape`. The next level of the hierarchy declares *concrete* classes for `TwoDimensionalShapes` (`Circle`, `Square` and `Triangle`) and for `ThreeDimensionalShapes` (`Sphere`, `Cube` and `Tetrahedron`).

Declaring an Abstract Class and Abstract Methods

You make a class abstract by declaring it with keyword **`abstract`**. An abstract class normally contains one or more **abstract methods**. An abstract method is one with keyword `abstract` in its declaration, as in

```
    public abstract void draw(); // abstract method
```

Abstract methods do *not* provide implementations. A class that contains *any* abstract methods must be explicitly declared `abstract` even if that class contains some concrete (nonabstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods. Constructors and `static` methods cannot be declared `abstract`. Constructors are not inherited, so an `abstract` constructor could never be implemented. Though non-`private static` methods *are* inherited, they *cannot* be overridden. Since `abstract` methods are meant to be overridden so that they can process objects based on their types, it would not make sense to declare a `static` method as `abstract`.

Software Engineering Observation G.6

An abstract class declares common attributes and behaviors (both abstract and concrete) of the various classes in a class hierarchy. An abstract class typically contains one or more abstract methods that subclasses must override if they are to be concrete. The instance variables and concrete methods of an abstract class are subject to the normal rules of inheritance.

Using Abstract Classes to Declare Variables

Although we cannot instantiate objects of abstract superclasses, you'll soon see that we *can* use abstract superclasses to declare variables that can hold references to objects of any concrete class derived from those abstract superclasses. Programs typically use such variables to manipulate subclass objects polymorphically. You also can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

Consider another application of polymorphism. A drawing program needs to display many shapes, including types of new shapes that you'll add to the system after writing the drawing program. The drawing program might need to display shapes, such as `Circles`, `Triangles`, `Rectangles` or others, that derive from abstract class `Shape`. The drawing program uses `Shape` variables to manage the objects that are displayed. To draw any object in this inheritance hierarchy, the drawing program uses a superclass `Shape` variable containing a reference to the subclass object to invoke the object's `draw` method. This method is declared `abstract` in superclass `Shape`, so each concrete subclass *must* implement method `draw` in a manner *specific* to that shape—each object in the `Shape` inheritance hierarchy *knows how to draw itself*. The drawing program does not have to worry about the type of each object or whether the program has ever encountered objects of that type.

G.10 Case Study: Payroll System Using Polymorphism

This section reexamines the `CommissionEmployee-BasePlusCommissionEmployee` hierarchy that we explored throughout Section G.4. Now we use an abstract method and polymorphism to perform payroll calculations based on an enhanced employee inheritance hierarchy that meets the following requirements:

A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants to write an application that performs its payroll calculations polymorphically.

We use `abstract` class `Employee` to represent the general concept of an employee. The classes that extend `Employee` are `SalariedEmployee`, `CommissionEmployee` and `HourlyEm-`

ployee. Class BasePlusCommissionEmployee—which extends CommissionEmployee—represents the last employee type. The UML class diagram in Fig. G.14 shows the inheritance hierarchy for our polymorphic employee-payroll application. Abstract class name Employee is italicized—a convention of the UML.

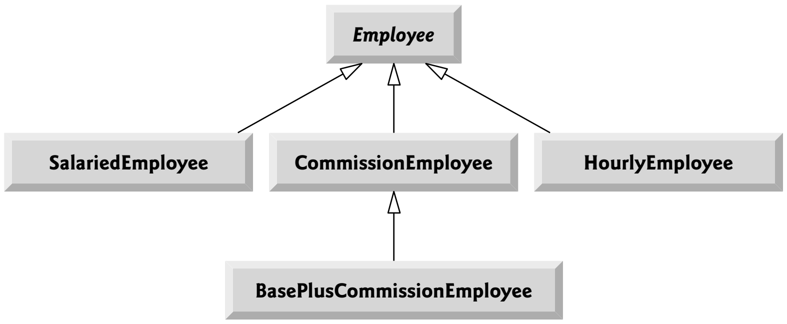


Fig. G.14 | Employee hierarchy UML class diagram.

Abstract superclass Employee declares the "interface" to the hierarchy—that is, the set of methods that a program can invoke on all Employee objects. We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any Employee subclass. Be careful not to confuse the general notion of an "interface" with the formal notion of a Java interface, the subject of Section G.12. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so private instance variables firstName, lastName and socialSecurityNumber appear in abstract superclass Employee.

The following sections implement the Employee class hierarchy of Fig. G.14. The first section implements abstract superclass Employee. The next four sections each implement one of the concrete classes. The last section implements a test program that builds objects of all these classes and processes those objects polymorphically.

G.10.1 Abstract Superclass Employee

Class Employee (Fig. G.16) provides methods earnings and toString, in addition to the *get* and *set* methods that manipulate Employee's instance variables. An earnings method certainly applies generically to all employees. But each earnings calculation depends on the employee's class. So we declare earnings as abstract in superclass Employee because a default implementation does not make sense for that method—there isn't enough information to determine what amount earnings should return. Each subclass overrides earnings with an appropriate implementation. To calculate an employee's earnings, the program assigns to a superclass Employee variable a reference to the employee's object, then invokes the earnings method on that variable. We maintain an array of Employee variables, each holding a reference to an Employee object. (Of course, there cannot be Employee objects, because Employee is an abstract class. Because of inheritance, however, all objects of all subclasses of Employee may nevertheless be thought of as Employee objects.) The program will iterate through the array and call method earnings for each Employee object. Java processes these method calls polymorphically. Declaring earnings as an abstract method in Employee enables the calls to earnings through Employee variables to compile and forces every direct concrete subclass of Employee to override earnings.

Method toString in class Employee returns a String containing the first name, last name and social security number of the employee. As we'll see, each subclass of Employee overrides method toString to create a String representation of an object of that class that contains the employee's type (e.g., "salaried employee:") followed by the rest of the employee's information.

The diagram in Fig. G.15 shows each of the five classes in the hierarchy down the left side and methods earnings and toString across the top. For each class, the diagram shows the desired results of each method. We do not list superclass Employee's *get* and *set* methods because they're not overridden in any of the subclasses—each of these methods is inherited and used "as is" by each subclass.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*
Hourly-Employee	if (hours <= 40) wage * hours else if (hours > 40) { 40 * wage + (hours - 40) * wage * 1.5 }	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. G.15 | Polymorphic interface for the Employee hierarchy classes.

Let's consider class Employee's declaration (Fig. G.16). The class includes a constructor that takes the first name, last name and social security number as arguments (lines 11–16); *get* methods that return the first name, last name and social security number (lines 25–28, 37–40 and 49–52, respectively); *set* methods that set the first name, last name and social security number (lines 19–22, 31–34 and 43–46, respectively); method toString (lines 55–60), which returns the `String` representation of an `Employee`; and `abstract` method `earnings` (line 63), which will be implemented by each of the concrete subclasses. The `Employee` constructor does not validate its parameters in this example; normally, such validation should be provided.

```
 1  // Fig. G.16: Employee.java
 2  // Employee abstract superclass.
 3
 4  public abstract class Employee
 5  {
 6     private String firstName;
 7     private String lastName;
 8     private String socialSecurityNumber;
 9
10     // three-argument constructor
11     public Employee( String first, String last, String ssn )
12     {
13        firstName = first;
14        lastName = last;
15        socialSecurityNumber = ssn;s
16     } // end three-argument Employee constructor
17
18     // set first name
19     public void setFirstName( String first )
20     {
21        firstName = first; // should validate
22     } // end method setFirstName
23
24     // return first name
25     public String getFirstName()
26     {
27        return firstName;
28     } // end method getFirstName
29
30     // set last name
31     public void setLastName( String last )
32     {
33        lastName = last; // should validate
34     } // end method setLastName
35
36     // return last name
37     public String getLastName()
38     {
39        return lastName;
40     } // end method getLastName
41
42     // set social security number
43     public void setSocialSecurityNumber( String ssn )
44     {
45        socialSecurityNumber = ssn; // should validate
46     } // end method setSocialSecurityNumber
47
```

Fig. G.16 | `Employee` abstract superclass. (Part 1 of 2.)

```
   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // return String representation of Employee object
   @Override
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // abstract method overridden by concrete subclasses
   public abstract double earnings(); // no implementation here
} // end abstract class Employee
```

Fig. G.16 | `Employee` abstract superclass. (Part 2 of 2.)

Why did we decide to declare `earnings` as an `abstract` method? It simply does not make sense to provide an implementation of this method in class `Employee`. We cannot calculate the earnings for a *general* `Employee`—we first must know the *specific* type of `Employee` to determine the appropriate earnings calculation. By declaring this method `abstract`, we indicate that each concrete subclass *must* provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

G.10.2 Concrete Subclass `SalariedEmployee`

Class `SalariedEmployee` (Fig. G.17) extends class `Employee` (line 4) and overrides abstract method `earnings` (lines 33–37), which makes `SalariedEmployee` a concrete class. The class includes a constructor (lines 9–14) that takes a first name, a last name, a social security number and a weekly salary as arguments; a *set* method to assign a new nonnegative value to instance variable `weeklySalary` (lines 17–24); a *get* method to return `weeklySalary`'s value (lines 27–30); a method `earnings` (lines 33–37) to calculate a `SalariedEmployee`'s earnings; and a method `toString` (lines 40–45), which returns a `String` including the employee's type, namely, `"salaried employee: "` followed by employee-specific information produced by superclass `Employee`'s `toString` method and `Salaried-Employee`'s `getWeeklySalary` method. Class `SalariedEmployee`'s constructor passes the first name, last name and social security number to the `Employee` constructor (line 12) to initialize the `private` instance variables not inherited from the superclass. Method `earnings` overrides `Employee`'s abstract method `earnings` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary. If we do not implement `earnings`, class `SalariedEmployee` must be declared `abstract`—otherwise, class `SalariedEmployee` will not compile. Of course, we want `SalariedEmployee` to be a concrete class in this example.

Method `toString` (lines 40–45) overrides `Employee` method `toString`. If class `SalariedEmployee` did not override `toString`, `SalariedEmployee` would have inherited the `Employee` version of `toString`. In that case, `SalariedEmployee`'s `toString` method would

```
 1  // Fig. G.17: SalariedEmployee.java
 2  // SalariedEmployee concrete class extends abstract class Employee.
 3
 4  public class SalariedEmployee extends Employee
 5  {
 6     private double weeklySalary;
 7
 8     // four-argument constructor
 9     public SalariedEmployee( String first, String last, String ssn,
10        double salary )
11     {
12        super( first, last, ssn ); // pass to Employee constructor
13        setWeeklySalary( salary ); // validate and store salary
14     } // end four-argument SalariedEmployee constructor
15
16     // set salary
17     public void setWeeklySalary( double salary )
18     {
19        if ( salary >= 0.0 )
20           baseSalary = salary;
21        else
22           throw new IllegalArgumentException(
23              "Weekly salary must be >= 0.0" );
24     } // end method setWeeklySalary
25
26     // return salary
27     public double getWeeklySalary()
28     {
29        return weeklySalary;
30     } // end method getWeeklySalary
31
32     // calculate earnings; override abstract method earnings in Employee
33     @Override
34     public double earnings()
35     {
36        return getWeeklySalary();
37     } // end method earnings
38
39     // return String representation of SalariedEmployee object
40     @Override
41     public String toString()
42     {
43        return String.format( "salaried employee: %s\n%s: $%,.2f",
44           super.toString(), "weekly salary", getWeeklySalary() );
45     } // end method toString
46  } // end class SalariedEmployee
```

Fig. G.17 | `SalariedEmployee` concrete class extends `abstract` class `Employee`.

simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To produce a complete `String` representation of a `SalariedEmployee`, the subclass's `toString` method returns `"salaried employee: "` followed by the superclass `Employee`-specific information (i.e., first name, last name and social security number) obtained by invoking the superclass's `toString` method (line

44)—this is a nice example of code reuse. The String representation of a SalariedEmployee also contains the employee's weekly salary obtained by invoking the class's getWeeklySalary method.

G.10.3 Concrete Subclass HourlyEmployee

Class HourlyEmployee (Fig. G.18) also extends Employee (line 4). The class includes a constructor (lines 10–16) that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked. Lines 19–26 and 35–42 declare *set* methods that assign new values to instance variables wage and hours, respectively. Method setWage (lines 19–26) ensures that wage is nonnegative, and method setHours (lines 35–42) ensures that hours is between 0 and 168 (the total number of hours in a week) inclusive. Class HourlyEmployee also includes *get* methods (lines 29–32 and 45–48) to return the values of wage and hours, respectively; a method earnings (lines 51–58) to calculate an HourlyEmployee's earnings; and a method toString (lines 61–67), which returns a String containing the employee's type ("hourly employee: ") and the employee-specific information. The HourlyEmployee constructor, like the SalariedEmployee constructor, passes the first name, last name and social security number to the superclass Employee constructor (line 13) to initialize the private instance variables. In addition, method toString calls superclass method toString (line 65) to obtain the Employee-specific information (i.e., first name, last name and social security number)—this is another nice example of code reuse.

```
 1  // Fig. G.18: HourlyEmployee.java
 2  // HourlyEmployee class extends Employee.
 3
 4  public class HourlyEmployee extends Employee
 5  {
 6     private double wage; // wage per hour
 7     private double hours; // hours worked for week
 8
 9     // five-argument constructor
10     public HourlyEmployee( String first, String last, String ssn,
11        double hourlyWage, double hoursWorked )
12     {
13        super( first, last, ssn );
14        setWage( hourlyWage ); // validate hourly wage
15        setHours( hoursWorked ); // validate hours worked
16     } // end five-argument HourlyEmployee constructor
17
18     // set wage
19     public void setWage( double hourlyWage )
20     {
21        if ( hourlyWage >= 0.0 )
22           wage = hourlyWage;
23        else
24           throw new IllegalArgumentException(
25              "Hourly wage must be >= 0.0" );
26     } // end method setWage
```

Fig. G.18 | HourlyEmployee class extends Employee. (Part 1 of 2.)

```java

   // return wage
   public double getWage()
   {
      return wage;
   } // end method getWage

   // set hours worked
   public void setHours( double hoursWorked )
   {
      if ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) )
         hours = hoursWorked;
      else
         throw new IllegalArgumentException(
            "Hours worked must be >= 0.0 and <= 168.0" );
   } // end method setHours

   // return hours worked
   public double getHours()
   {
      return hours;
   } // end method getHours

   // calculate earnings; override abstract method earnings in Employee
   @Override
   public double earnings()
   {
      if ( getHours() <= 40 ) // no overtime
         return getWage() * getHours();
      else
         return 40 * getWage() + ( getHours() - 40 ) * getWage() * 1.5;
   } // end method earnings

   // return String representation of HourlyEmployee object
   @Override
   public String toString()
   {
      return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
         super.toString(), "hourly wage", getWage(),
         "hours worked", getHours() );
   } // end method toString
} // end class HourlyEmployee
```

Fig. G.18 | `HourlyEmployee` class extends `Employee`. (Part 2 of 2.)

G.10.4 Concrete Subclass `CommissionEmployee`

Class `CommissionEmployee` (Fig. G.19) extends class `Employee` (line 4). The class includes a constructor (lines 10–16) that takes a first name, a last name, a social security number, a sales amount and a commission rate; *set* methods (lines 19–26 and 35–42) to assign new values to instance variables `commissionRate` and `grossSales`, respectively; *get* methods (lines 29–32 and 45–48) that retrieve the values of these instance variables; method `earnings` (lines 51–55) to calculate a `CommissionEmployee`'s earnings; and method `toString` (lines 58–65), which returns the employee's type, namely, `"commission employee: "` and employee-specific information. The constructor also passes the first name, last name and social security number to `Employee`'s constructor (line 13) to initialize `Employee`'s `private` instance variables. Method `toString` calls superclass method `toString` (line 62) to obtain the `Employee`-specific information (i.e., first name, last name and social security number).

```java
 1  // Fig. G.19: CommissionEmployee.java
 2  // CommissionEmployee class extends Employee.
 3
 4  public class CommissionEmployee extends Employee
 5  {
 6     private double grossSales; // gross weekly sales
 7     private double commissionRate; // commission percentage
 8
 9     // five-argument constructor
10     public CommissionEmployee( String first, String last, String ssn,
11        double sales, double rate )
12     {
13        super( first, last, ssn );
14        setGrossSales( sales );
15        setCommissionRate( rate );
16     } // end five-argument CommissionEmployee constructor
17
18     // set commission rate
19     public void setCommissionRate( double rate )
20     {
21        if ( rate > 0.0 && rate < 1.0 )
22           commissionRate = rate;
23        else
24           throw new IllegalArgumentException(
25              "Commission rate must be > 0.0 and < 1.0" );
26     } // end method setCommissionRate
27
28     // return commission rate
29     public double getCommissionRate()
30     {
31        return commissionRate;
32     } // end method getCommissionRate
33
34     // set gross sales amount
35     public void setGrossSales( double sales )
36     {
37        if ( sales >= 0.0 )
38           grossSales = sales;
39        else
40           throw new IllegalArgumentException(
41              "Gross sales must be >= 0.0" );
42     } // end method setGrossSales
43
44     // return gross sales amount
45     public double getGrossSales()
46     {
```

Fig. G.19 | `CommissionEmployee` class extends `Employee`. (Part 1 of 2.)

```
         return grossSales;
      } // end method getGrossSales

      // calculate earnings; override abstract method earnings in Employee
      @Override
      public double earnings()
      {
         return getCommissionRate() * getGrossSales();
      } // end method earnings

      // return String representation of CommissionEmployee object
      @Override
      public String toString()
      {
         return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
            "commission employee", super.toString(),
            "gross sales", getGrossSales(),
            "commission rate", getCommissionRate() );
      } // end method toString
   } // end class CommissionEmployee
```

Fig. G.19 | `CommissionEmployee` class extends `Employee`. (Part 2 of 2.)

G.10.5 Indirect Concrete Subclass `BasePlusCommissionEmployee`

Class `BasePlusCommissionEmployee` (Fig. G.20) extends class `CommissionEmployee` (line 4) and therefore is an *indirect* subclass of class `Employee`. Class `BasePlusCommissionEmployee` has a constructor (lines 9–14) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes all of these except the base salary to the `CommissionEmployee` constructor (line 12) to initialize the inherited members. `BasePlusCommissionEmployee` also contains a *set* method (lines 17–24) to assign a new value to instance variable `baseSalary` and a *get* method (lines 27–30) to return `baseSalary`'s value. Method `earnings` (lines 33–37) calculates a `BasePlusCommissionEmployee`'s earnings. Line 36 in method `earnings` calls superclass `CommissionEmployee`'s `earnings` method to calculate the commission-based portion of the employee's earnings—this is another nice example of code reuse. `BasePlusCommissionEmployee`'s `toString` method (lines 40–46) creates a `String` representation of a `BasePlusCommissionEmployee` that contains `"base-salaried"`, followed by the `String`

```
 1   // Fig. G.20: BasePlusCommissionEmployee.java
 2   // BasePlusCommissionEmployee class extends CommissionEmployee.
 3
 4   public class BasePlusCommissionEmployee extends CommissionEmployee
 5   {
 6      private double baseSalary; // base salary per week
 7
 8      // six-argument constructor
 9      public BasePlusCommissionEmployee( String first, String last,
10         String ssn, double sales, double rate, double salary )
11      {
```

Fig. G.20 | `BasePlusCommissionEmployee` class extends `CommissionEmployee`. (Part 1 of 2.)

```
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0" );
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings; override method earnings in CommissionEmployee
   @Override
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee object
   @Override
   public String toString()
   {
      return String.format( "%s %s; %s: $%,.2f",
         "base-salaried", super.toString(),
         "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

Fig. G.20 | `BasePlusCommissionEmployee` class extends `CommissionEmployee`. (Part 2 of 2.)

obtained by invoking superclass `CommissionEmployee`'s `toString` method (another example of code reuse), then the base salary. The result is a `String` beginning with `"base-salaried commission employee"` followed by the rest of the `BasePlusCommissionEmployee`'s information. Recall that `CommissionEmployee`'s `toString` obtains the employee's first name, last name and social security number by invoking the `toString` method of its superclass (i.e., `Employee`)—yet another example of code reuse. `BasePlusCommissionEmployee`'s `toString` initiates a chain of method calls that span all three levels of the `Employee` hierarchy.

G.10.6 Polymorphic Processing, Operator `instanceof` and Downcasting

To test our `Employee` hierarchy, the application in Fig. G.21 creates an object of each of the four concrete classes `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`. The program manipulates these objects nonpolymorphically, via variables of each object's own type, then polymorphically, using an array of `Employee` variables. While processing the objects polymorphically, the program increases the base salary of each `BasePlusCommissionEmployee` by 10%—this requires *determining the object's type at execution time*. Finally, the program polymorphically determines and outputs the type of each object in the `Employee` array. Lines 9–18 create objects of each of the four concrete `Employee` subclasses. Lines 22–30 output the `String` representation and earnings of each of these objects *nonpolymorphically*. Each object's `toString` method is called *implicitly* by `printf` when the object is output as a `String` with the `%s` format specifier.

```
 1  // Fig. G.21: PayrollSystemTest.java
 2  // Employee hierarchy test program.
 3
 4  public class PayrollSystemTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // create subclass objects
 9        SalariedEmployee salariedEmployee =
10           new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11        HourlyEmployee hourlyEmployee =
12           new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13        CommissionEmployee commissionEmployee =
14           new CommissionEmployee(
15           "Sue", "Jones", "333-33-3333", 10000, .06 );
16        BasePlusCommissionEmployee basePlusCommissionEmployee =
17           new BasePlusCommissionEmployee(
18           "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20        System.out.println( "Employees processed individually:\n" );
21
22        System.out.printf( "%s\n%s: $%,.2f\n\n",
23           salariedEmployee, "earned", salariedEmployee.earnings() );
24        System.out.printf( "%s\n%s: $%,.2f\n\n",
25           hourlyEmployee, "earned", hourlyEmployee.earnings() );
26        System.out.printf( "%s\n%s: $%,.2f\n\n",
27           commissionEmployee, "earned", commissionEmployee.earnings() );
28        System.out.printf( "%s\n%s: $%,.2f\n\n",
29           basePlusCommissionEmployee,
30           "earned", basePlusCommissionEmployee.earnings() );
31
32        // create four-element Employee array
33        Employee[] employees = new Employee[ 4 ];
34
35        // initialize array with Employees
36        employees[ 0 ] = salariedEmployee;
37        employees[ 1 ] = hourlyEmployee;
38        employees[ 2 ] = commissionEmployee;
39        employees[ 3 ] = basePlusCommissionEmployee;
40
41        System.out.println( "Employees processed polymorphically:\n" );
42
```

Fig. G.21 | `Employee` hierarchy test program. (Part 1 of 3.)

```
      // generically process each element in array employees
      for ( Employee currentEmployee : employees )
      {
         System.out.println( currentEmployee ); // invokes toString

         // determine whether element is a BasePlusCommissionEmployee
         if ( currentEmployee instanceof BasePlusCommissionEmployee )
         {
            // downcast Employee reference to
            // BasePlusCommissionEmployee reference
            BasePlusCommissionEmployee employee =
               ( BasePlusCommissionEmployee ) currentEmployee;

            employee.setBaseSalary( 1.10 * employee.getBaseSalary() );

            System.out.printf(
               "new base salary with 10%% increase is: $%,.2f\n",
               employee.getBaseSalary() );
         } // end if

         System.out.printf(
            "earned $%,.2f\n\n", currentEmployee.earnings() );
      } // end for

      // get type name of each object in employees array
      for ( int j = 0; j < employees.length; j++ )
         System.out.printf( "Employee %d is a %s\n", j,
            employees[ j ].getClass().getName() );
   } // end main
} // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00
```

Fig. G.21 | `Employee` hierarchy test program. (Part 2 of 3.)

```
Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Fig. G.21 | `Employee` hierarchy test program. (Part 3 of 3.)

Creating the Array of `Employee`*s*

Line 33 declares `employees` and assigns it an array of four `Employee` variables. Line 36 assigns the reference to a `SalariedEmployee` object to `employees[0]`. Line 37 assigns the reference to an `HourlyEmployee` object to `employees[1]`. Line 38 assigns the reference to a `CommissionEmployee` object to `employees[2]`. Line 39 assigns the reference to a `BasePlusCommissionEmployee` object to `employee[3]`. These assignments are allowed, because a `SalariedEmployee` *is an* `Employee`, an `HourlyEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`. Therefore, we can assign the references of `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee` objects to superclass `Employee` variables, *even though* `Employee` *is an abstract class*.

Polymorphically Processing `Employee`*s*

Lines 44–65 iterate through array `employees` and invoke methods `toString` and `earnings` with `Employee` variable `currentEmployee`, which is assigned the reference to a different `Employee` in the array on each iteration. The output illustrates that the appropriate methods for each class are indeed invoked. All calls to method `toString` and `earnings` are resolved at execution time, based on the type of the object to which `currentEmployee` refers. This process is known as **dynamic binding** or **late binding**. For example, line 46 *implicitly* invokes method `toString` of the object to which `currentEmployee` refers. As a result of dynamic binding, Java decides which class's `toString` method to call *at execution time rather than at compile time*. Only the methods of class `Employee` can be called via an `Em-`

`ployee` variable (and `Employee`, of course, includes the methods of class `Object`). A superclass reference can be used to invoke only methods of the superclass—the subclass method implementations are invoked polymorphically.

Performing Type-Specific Operations on **`BasePlusCommissionEmployee`***s*

We perform special processing on `BasePlusCommissionEmployee` objects—as we encounter these objects at execution time, we increase their base salary by 10%. When processing objects polymorphically, we typically do not need to worry about the "specifics," but to adjust the base salary, we *do* have to determine the specific type of `Employee` object at execution time. Line 49 uses the **`instanceof`** operator to determine whether a particular `Employee` object's type is `BasePlusCommissionEmployee`. The condition in line 49 is true if the object referenced by `currentEmployee` *is a* `BasePlusCommissionEmployee`. This would also be true for any object of a `BasePlusCommissionEmployee` subclass because of the *is-a* relationship a subclass has with its superclass. Lines 53–54 downcast `currentEmployee` from type `Employee` to type `BasePlusCommissionEmployee`—this cast is allowed only if the object has an *is-a* relationship with `BasePlusCommissionEmployee`. The condition at line 49 ensures that this is the case. This cast is required if we're to invoke subclass `BasePlusCommissionEmployee` methods `getBaseSalary` and `setBaseSalary` on the current `Employee` object—as you'll see momentarily, *attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.*

Common Programming Error G.3

Assigning a superclass variable to a subclass variable (without an explicit cast) is a compilation error.

Software Engineering Observation G.7

If a subclass object's reference has been assigned to a variable of one of its direct or indirect superclasses at execution time, it's acceptable to downcast the reference stored in that superclass variable back to a subclass-type reference. Before performing such a cast, use the `instanceof` operator to ensure that the object is indeed an object of an appropriate subclass.

Common Programming Error G.4

When downcasting a reference, a `ClassCastException` occurs if the referenced object at execution time does not have an is-a *relationship with the type specified in the cast operator.*

If the `instanceof` expression in line 49 is `true`, lines 53–60 perform the special processing required for the `BasePlusCommissionEmployee` object. Using `BasePlusCommissionEmployee` variable `employee`, line 56 invokes subclass-only methods `getBaseSalary` and `setBaseSalary` to retrieve and update the employee's base salary with the 10% raise.

Calling **`earnings`** *Polymorphically*

Lines 63–64 invoke method `earnings` on `currentEmployee`, which polymorphically calls the appropriate subclass object's `earnings` method. Obtaining the earnings of the `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` polymorphically in lines 63–64 produces the same results as obtaining these employees' earnings individually in lines 22–27. The earnings amount obtained for the `BasePlusCommissionEmployee` in lines 63–64 is higher than that obtained in lines 28–30, due to the 10% increase in its base salary.

Using Reflection to Get Each **`Employee`***'s Class Name*

Lines 68–70 display each employee's type as a `String`, using basic features of Java's so-called reflection capabilities. Every object knows its own class and can access this information through the **`getClass`** method, which all classes inherit from class `Object`. Method `getClass` returns an object of type **`Class`** (from package `java.lang`), which contains information about the object's type, including its class name. Line 70 invokes `getClass` on the current object to get its runtime class. The result of the `getClass` call is used to invoke **`getName`** to get the object's class name.

Avoiding Compilation Errors with Downcasting

In the previous example, we avoided several compilation errors by downcasting an `Employee` variable to a `BasePlusCommissionEmployee` variable in lines 53–54. If you remove the cast operator `(BasePlusCommissionEmployee)` from line 54 and attempt to assign `Employee` variable `currentEmployee` directly to `BasePlusCommissionEmployee` variable `employee`, you'll receive an "`incompatible types`" compilation error. This error indicates that the attempt to assign the reference of superclass object `currentEmployee` to subclass variable `employee` is not allowed. The compiler prevents this assignment because a `CommissionEmployee` is not a `BasePlusCommissionEmployee`—*the* is-a *relationship applies only between the subclass and its superclasses, not vice versa.*

Similarly, if lines 56 and 60 used superclass variable `currentEmployee` to invoke subclass-only methods `getBaseSalary` and `setBaseSalary`, we'd receive "`cannot find symbol`" compilation errors at these lines. Attempting to invoke subclass-only methods via a superclass variable is not allowed—even though lines 56 and 60 execute only if `instanceof` in line 49 returns `true` to indicate that `currentEmployee` holds a reference to a `BasePlusCommissionEmployee` object. Using a superclass `Employee` variable, we can invoke only methods found in class `Employee`—`earnings`, `toString` and `Employee`'s *get* and *set* methods.

Software Engineering Observation G.8

Although the actual method that's called depends on the runtime type of the object to which a variable refers, a variable can be used to invoke only those methods that are members of that variable's type, which the compiler verifies.

G.10.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables

Now that you've seen a complete application that processes diverse subclass objects polymorphically, we summarize what you can and cannot do with superclass and subclass objects and variables. Although a subclass object also *is a* superclass object, the two objects are nevertheless different. As discussed previously, subclass objects can be treated as objects of their superclass. But because the subclass can have additional subclass-only members, assigning a superclass reference to a subclass variable is not allowed without an explicit cast—such an assignment would leave the subclass members undefined for the superclass object.

We've discussed four ways to assign superclass and subclass references to variables of superclass and subclass types:

1. Assigning a superclass reference to a superclass variable is straightforward.
2. Assigning a subclass reference to a subclass variable is straightforward.

3. Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an* object of its superclass. However, the superclass variable can be used to refer *only* to superclass members. If this code refers to subclass-only members through the superclass variable, the compiler reports errors.
4. Attempting to assign a superclass reference to a subclass variable is a compilation error. To avoid this error, the superclass reference must be cast to a subclass type explicitly. At *execution time*, if the object to which the reference refers is *not* a subclass object, an exception will occur. (For more on exception handling, see Appendix H.) You should use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.

G.11 `final` Methods and Classes

We saw in Sections D.3 and D.10 that variables can be declared `final` to indicate that they cannot be modified after they're initialized—such variables represent constant values. It's also possible to declare methods, method parameters and classes with the `final` modifier.

Final Methods Cannot Be Overridden

A **`final` method** in a superclass *cannot* be overridden in a subclass—this guarantees that the `final` method implementation will be used by all direct and indirect subclasses in the hierarchy. Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass. Methods that are declared `static` are also implicitly `final`. A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as **static binding**.

Final Classes Cannot Be Superclasses

A **`final` class** that's declared `final` cannot be a superclass (i.e., a class cannot extend a `final` class). All methods in a `final` class are implicitly `final`. Class `String` is an example of a `final` class. If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected. Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API. Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions. For more insights on the use of keyword `final`, visit

```
docs.oracle.com/javase/tutorial/java/IandI/final.html
```

and

```
www.ibm.com/developerworks/java/library/j-jtp1029.html
```

Common Programming Error G.5

Attempting to declare a subclass of a `final` class is a compilation error.

Software Engineering Observation G.9

In the Java API, the vast majority of classes are not *declared `final`. This enables inheritance and polymorphism. However, in some cases, it's important to declare classes `final`—typically for security reasons.*

G.12 Case Study: Creating and Using Interfaces

Our next example (Figs. G.23–G.27) reexamines the payroll system of Section G.10. Suppose that the company involved wishes to perform several accounting operations in a single accounts payable application—in addition to calculating the earnings that must be paid to each employee, the company must also calculate the payment due on each of several invoices (i.e., bills for goods purchased). Though applied to unrelated things (i.e., employees and invoices), both operations have to do with obtaining some kind of payment amount. For an employee, the payment refers to the employee's earnings. For an invoice, the payment refers to the total cost of the goods listed on the invoice. Can we calculate such *different* things as the payments due for employees and invoices in *a single* application polymorphically? Does Java offer a capability requiring that *unrelated* classes implement a set of *common* methods (e.g., a method that calculates a payment amount)? Java **interfaces** offer exactly this capability.

Standardizing Interactions

Interfaces define and standardize the ways in which things such as people and systems can interact with one another. For example, the controls on a radio serve as an interface between radio users and a radio's internal components. The controls allow users to perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM), and different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands). The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.

Software Objects Communicate Via Interfaces

Software objects also communicate via interfaces. A Java interface describes a set of methods that can be called on an object to tell it, for example, to perform some task or return some piece of information. The next example introduces an interface named `Payable` to describe the functionality of any object that must be capable of being paid and thus must offer a method to determine the proper payment amount due. An **interface declaration** begins with the keyword **`interface`** and contains only constants and `abstract` methods. Unlike classes, all interface members must be `public`, and *interfaces may not specify any implementation details*, such as concrete method declarations and instance variables. All methods declared in an interface are implicitly `public abstract` methods, and all fields are implicitly `public`, `static` and `final`. [*Note:* As of Java SE 5, it became a better programming practice to declare sets of constants as enumerations with keyword `enum`. See Section D.10 for an introduction to `enum` and Section F.8 for additional `enum` details.]

Good Programming Practice G.1

According to Chapter 9 of the Java Language Specification*, it's proper style to declare an interface's methods without keywords `public` and `abstract`, because they're redundant in interface method declarations. Similarly, constants should be declared without keywords `public`, `static` and `final`, because they, too, are redundant.*

Using an Interface

To use an interface, a concrete class must specify that it **`implements`** the interface and must declare each method in the interface with the signature specified in the interface declaration. To specify that a class implements an interface add the `implements` keyword and the name of the interface to the end of your class declaration's first line. A class that does not implement *all* the methods of the interface is an *abstract* class and must be declared `abstract`. Implementing an interface is like signing a *contract* with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class `abstract`."

Common Programming Error G.6

Failing to implement any method of an interface in a concrete class that `implements` the interface results in a compilation error indicating that the class must be declared `abstract`.

Relating Disparate Types

An interface is often used when disparate (i.e., unrelated) classes need to share common methods and constants. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to the same method calls. You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality. For example, in the accounts payable application developed in this section, we implement interface `Payable` in any class that must be able to calculate a payment amount (e.g., `Employee`, `Invoice`).

Interfaces vs. Abstract Classes

An interface is often used in place of an `abstract` class when there's no default implementation to inherit—that is, no fields and no default method implementations. Like `public abstract` classes, interfaces are typically `public` types. Like a `public` class, a `public` interface must be declared in a file with the same name as the interface and the `.java` file-name extension.

Tagging Interfaces

We'll see in Appendix J, the notion of "tagging interfaces"—empty interfaces that have *no* methods or constant values. They're used to add *is-a* relationships to classes. For example, in Appendix J we'll discuss a mechanism called object serialization, which can convert objects to byte representations and can convert those byte representations back to objects. To enable this mechanism to work with your objects, you simply have to mark them as `Serializable` by adding `implements Serializable` to the end of your class declaration's first line. Then, all the objects of your class have the *is-a* relationship with `Serializable`.

G.12.1 Developing a `Payable` Hierarchy

To build an application that can determine payments for employees and invoices alike, we first create interface `Payable`, which contains method `getPaymentAmount` that returns a `double` amount that must be paid for an object of any class that implements the interface. Method `getPaymentAmount` is a general-purpose version of method `earnings` of the `Employee` hierarchy—method `earnings` calculates a payment amount specifically for an `Employee`, while `getPaymentAmount` can be applied to a broad range of unrelated objects. After declaring interface `Payable`, we introduce class `Invoice`, which `implements` interface `Payable`. We then modify class `Employee` such that it also implements interface `Payable`. Finally, we update `Employee` subclass `SalariedEmployee` to "fit" into the `Payable` hierarchy by renaming `SalariedEmployee` method `earnings` as `getPaymentAmount`.

Good Programming Practice G.2

When declaring a method in an interface, choose a method name that describes the method's purpose in a general *manner, because the method may be implemented by many unrelated classes.*

Classes `Invoice` and `Employee` both represent things for which the company must be able to calculate a payment amount. Both classes implement the `Payable` interface, so a program can invoke method `getPaymentAmount` on `Invoice` objects and `Employee` objects alike. As we'll soon see, this enables the polymorphic processing of `Invoices` and `Employees` required for the company's accounts payable application.

The UML class diagram in Fig. G.22 shows the hierarchy used in our accounts payable application. The hierarchy begins with interface `Payable`. The UML distinguishes an interface from other classes by placing the word "interface" in guillemets (« and ») above the interface name. The UML expresses the relationship between a class and an interface through a relationship known as **realization**. A class is said to "realize," or implement, the methods of an interface. A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface. The diagram in Fig. G.22 indicates that classes `Invoice` and `Employee` each realize (i.e., implement) interface `Payable`. As in the class diagram of Fig. G.14, class `Employee` appears in italics, indicating that it's an abstract class. Concrete class `SalariedEmployee` extends `Employee` and *inherits its superclass's realization relationship* with interface `Payable`.

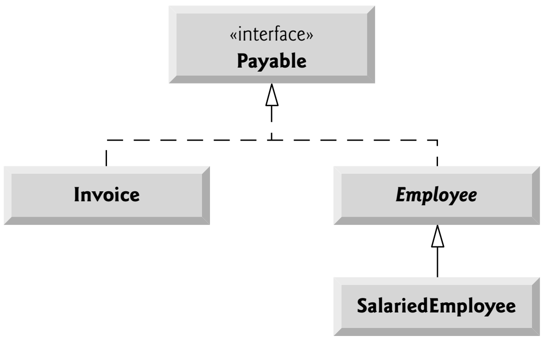


Fig. G.22 | `Payable` interface hierarchy UML class diagram.

G.12.2 Interface `Payable`

The declaration of interface `Payable` begins in Fig. G.23 at line 4. Interface `Payable` contains `public abstract` method `getPaymentAmount` (line 6). The method is not explicitly declared `public` or `abstract`. Interface methods are always `public` and `abstract`, so they do not need to be declared as such. Interface `Payable` has only one method—interfaces can have any number of methods. In addition, method `getPaymentAmount` has no parameters, but interface methods *can* have parameters. Interfaces may also contain fields that are implicitly `final` and `static`.

```java
1  // Fig. G.23: Payable.java
2  // Payable interface declaration.
3
4  public interface Payable
5  {
6     double getPaymentAmount(); // calculate payment; no implementation
7  } // end interface Payable
```

Fig. G.23 | `Payable` interface declaration.

G.12.3 Class Invoice

We now create class `Invoice` (Fig. G.24) to represent a simple invoice that contains billing information for only one kind of part. The class declares `private` instance variables `partNumber`, `partDescription`, `quantity` and `pricePerItem` (in lines 6–9) that indicate the part number, a description of the part, the quantity of the part ordered and the price per item. Class `Invoice` also contains a constructor (lines 12–19), *get* and *set* methods (lines 22–74) that manipulate the class's instance variables and a `toString` method (lines 77–83) that returns a `String` representation of an `Invoice` object. Methods `setQuantity` (lines 46–52) and `setPricePerItem` (lines 61–68) ensure that `quantity` and `pricePerItem` obtain only nonnegative values.

```java
1  // Fig. G.24: Invoice.java
2  // Invoice class that implements Payable.
3
4  public class Invoice implements Payable
5  {
6     private String partNumber;
7     private String partDescription;
8     private int quantity;2
9     private double pricePerItem;
10
11    // four-argument constructor
12    public Invoice( String part, String description, int count,
13       double price )
14    {
15       partNumber = part;
16       partDescription = description;
17       setQuantity( count ); // validate and store quantity
18       setPricePerItem( price ); // validate and store price per item
19    } // end four-argument Invoice constructor
20
21    // set part number
22    public void setPartNumber( String part )
23    {
24       partNumber = part; // should validate
25    } // end method setPartNumber
26
```

Fig. G.24 | `Invoice` class that implements `Payable`. (Part 1 of 3.)

```java
   // get part number
   public String getPartNumber()
   {
      return partNumber;
   } // end method getPartNumber

   // set description
   public void setPartDescription( String description )
   {
      partDescription = description; // should validate
   } // end method setPartDescription

   // get description
   public String getPartDescription()
   {
      return partDescription;
   } // end method getPartDescription

   // set quantity
   public void setQuantity( int count )
   {
      if ( count >= 0 )
         quantity = count;
      else
         throw new IllegalArgumentException( "Quantity must be >= 0" );
   } // end method setQuantity

   // get quantity
   public int getQuantity()
   {
      return quantity;
   } // end method getQuantity

   // set price per item
   public void setPricePerItem( double price )
   {
      if ( price >= 0.0 )
         pricePerItem = price;
      else
         throw new IllegalArgumentException(
            "Price per item must be >= 0" );
   } // end method setPricePerItem

   // get price per item
   public double getPricePerItem()
   {
      return pricePerItem;
   } // end method getPricePerItem

   // return String representation of Invoice object
   @Override
   public String toString()
   {
```

Fig. G.24 | `Invoice` class that implements `Payable`. (Part 2 of 3.)

```
80          return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
81             "invoice", "part number", getPartNumber(), getPartDescription(),
82             "quantity", getQuantity(), "price per item", getPricePerItem() );
83       } // end method toString
84
85       // method required to carry out contract with interface Payable
86       @Override
87       public double getPaymentAmount()
88       {
89          return getQuantity() * getPricePerItem(); // calculate total cost
90       } // end method getPaymentAmount
91    } // end class Invoice
```

Fig. G.24 | `Invoice` class that implements `Payable`. (Part 3 of 3.)

Line 4 indicates that class `Invoice` implements interface `Payable`. Like all classes, class `Invoice` also implicitly extends `Object`. Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs. To implement more than one interface, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class ClassName extends SuperclassName implements FirstInterface,
   SecondInterface, …
```

Software Engineering Observation G.10

All objects of a class that implement multiple interfaces have the is-a *relationship with each implemented interface type.*

Class `Invoice` implements the one method in interface `Payable`—method `getPaymentAmount` is declared in lines 86–90. The method calculates the total payment required to pay the invoice. The method multiplies the values of `quantity` and `pricePerItem` (obtained through the appropriate *get* methods) and returns the result (line 89). This method satisfies the implementation requirement for this method in interface `Payable`—we've fulfilled the interface contract with the compiler.

G.12.4 Modifying Class `Employee` to Implement Interface `Payable`

We now modify class `Employee` such that it implements interface `Payable`. Figure G.25 contains the modified class, which is identical to that of Fig. G.16 with two exceptions. First, line 4 of Fig. G.25 indicates that class `Employee` now `implements` interface `Payable`. So we must rename `earnings` to `getPaymentAmount` throughout the `Employee` hierarchy. As with method `earnings` in the version of class `Employee` in Fig. G.16, however, it does not make sense to implement method `getPaymentAmount` in class `Employee` because we cannot calculate the earnings payment owed to a general `Employee`—we must first know the specific type of `Employee`. In Fig. G.16, we declared method `earnings` as `abstract` for this reason, so class `Employee` had to be declared `abstract`. This forced each `Employee` concrete subclass to override `earnings` with an implementation.

In Fig. G.25, we handle this situation differently. Recall that when a class implements an interface, it makes a *contract* with the compiler stating either that the class will implement *each* of the methods in the interface or that the class will be declared `abstract`. If the

```java
// Fig. G.25: Employee.java
// Employee abstract superclass that implements Payable.

public abstract class Employee implements Payable
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last; // should validate
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

```

Fig. G.25 | `Employee` `abstract` superclass that implements `Payable`. (Part 1 of 2.)

```
54      // return String representation of Employee object
55      @Override
56      public String toString()
57      {
58         return String.format( "%s %s\nsocial security number: %s",
59            getFirstName(), getLastName(), getSocialSecurityNumber() );
60      } // end method toString
61
62      // Note: We do not implement Payable method getPaymentAmount here so
63      // this class must be declared abstract to avoid a compilation error.
64   } // end abstract class Employee
```

Fig. G.25 | `Employee` `abstract` superclass that implements `Payable`. (Part 2 of 2.)

latter option is chosen, we do not need to declare the interface methods as `abstract` in the `abstract` class—they're already implicitly declared as such in the interface. Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the superclass's contract with the compiler. If the subclass does not do so, it too must be declared `abstract`. As indicated by the comments in lines 62–63, class `Employee` of Fig. G.25 does *not* implement method `getPaymentAmount`, so the class is declared `abstract`. Each direct `Employee` subclass *inherits the superclass's contract* to implement method `getPaymentAmount` and thus must implement this method to become a concrete class for which objects can be instantiated. A class that extends one of `Employee`'s concrete subclasses will inherit an implementation of `getPaymentAmount` and thus will also be a concrete class.

G.12.5 Modifying Class `SalariedEmployee` for Use in the `Payable` Hierarchy

Figure G.26 contains a modified `SalariedEmployee` class that extends `Employee` and fulfills superclass `Employee`'s contract to implement `Payable` method `getPaymentAmount`. This version of `SalariedEmployee` is identical to that of Fig. G.17, but it replaces method `earnings` with method `getPaymentAmount` (lines 34–38). Recall that the `Payable` version of the method has a more *general* name to be applicable to possibly *disparate* classes. The remaining `Employee` subclasses (e.g., `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`) also must be modified to contain method `getPaymentAmount` in place of `earnings` to reflect the fact that `Employee` now implements `Payable`. We leave these modifications as an exercise (Exercise G.16) and use only `SalariedEmployee` in our test program here. Exercise G.17 asks you to implement interface `Payable` in the entire `Employee` class hierarchy of Figs. G.16–G.21 without modifying the `Employee` subclasses.

When a class implements an interface, the same *is-a* relationship provided by inheritance applies. Class `Employee` implements `Payable`, so we can say that an `Employee` *is a* `Payable`. In fact, objects of any classes that extend `Employee` are also `Payable` objects. `SalariedEmployee` objects, for instance, are `Payable` objects. Objects of any subclasses of the class that `implements` the interface can also be thought of as objects of the interface type. Thus, just as we can assign the reference of a `SalariedEmployee` object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object to an interface `Payable` variable. `Invoice implements Payable`, so an `Invoice` object also *is a* `Payable` object, and we can assign the reference of an `Invoice` object to a `Payable` variable.

Software Engineering Observation G.11

When a method parameter is declared with a superclass or interface type, the method processes the object received as an argument polymorphically.

Software Engineering Observation G.12

Using a superclass reference, we can polymorphically invoke any method declared in the superclass and its superclasses (e.g., class `Object`*). Using an interface reference, we can polymorphically invoke any method declared in the interface, its superinterfaces (one interface can extend another) and in class* `Object`*—a variable of an interface type must refer to an object to call methods, and all objects have the methods of class* `Object`*.*

```java
// Fig. G.26: SalariedEmployee.java
// SalariedEmployee class extends Employee, which implements Payable.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Weekly salary must be >= 0.0" );
   } // end method setWeeklySalary

   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; implement interface Payable method that was
   // abstract in superclass Employee
   @Override
   public double getPaymentAmount()
   {
      return getWeeklySalary();
   } // end method getPaymentAmount
```

Fig. G.26 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 1 of 2.)

```

   // return String representation of SalariedEmployee object
   @Override
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Fig. G.26 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 2 of 2.)

G.12.6 Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically

`PayableInterfaceTest` (Fig. G.27) illustrates that interface `Payable` can be used to process a set of `Invoices` and `Employees` polymorphically in a single application. Line 9 declares `payableObjects` and assigns it an array of four `Payable` variables. Lines 12–13 assign the references of `Invoice` objects to the first two elements of `payableObjects`. Lines 14–17 then assign the references of `SalariedEmployee` objects to the remaining two elements of `payableObjects`. These assignments are allowed because an `Invoice` *is a* `Payable`, a `SalariedEmployee` *is an* `Employee` and an `Employee` *is a* `Payable`. Lines 23–29 use the enhanced `for` statement to polymorphically process each `Payable` object in `payableObjects`, printing the object as a `String`, along with the payment amount due. Line 27 invokes method `toString` via a `Payable` interface reference, even though `toString` is not declared in interface `Payable`—*all references (including those of interface types) refer to objects that extend `Object` and therefore have a `toString` method.* (Method `toString` also can be invoked *implicitly* here.) Line 28 invokes `Payable` method `getPaymentAmount` to obtain the payment amount for each object in `payableObjects`, regardless of the actual type of the object. The output reveals that the method calls in lines 27–28 invoke the appropriate class's implementation of methods `toString` and `getPaymentAmount`. For instance, when `currentPayable` refers to an `Invoice` during the first iteration of the `for` loop, class `Invoice`'s `toString` and `getPaymentAmount` execute.

```
 1  // Fig. G.27: PayableInterfaceTest.java
 2  // Tests interface Payable.
 3
 4  public class PayableInterfaceTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // create four-element Payable array
 9        Payable[] payableObjects = new Payable[ 4 ];
10
11        // populate array with objects that implement Payable
12        payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
```

Fig. G.27 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 1 of 2.)

```
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Invoices and Employees processed polymorphically:\n" );

      // generically process each element in array payableObjects
      for ( Payable currentPayable : payableObjects )
      {
         // output currentPayable and its appropriate payment amount
         System.out.printf( "%s \n%s: $%,.2f\n\n",
            currentPayable.toString(),
            "payment due", currentPayable.getPaymentAmount() );
      } // end for
   } // end main
} // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. G.27 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 2 of 2.)

G.13 Common Interfaces of the Java API

In this section, we overview several common interfaces found in the Java API. The power and flexibility of interfaces is used frequently throughout the Java API. These interfaces are implemented and used in the same manner as the interfaces you create (e.g., interface `Payable` in Section G.12.2). The Java API's interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program. Figure G.28 overviews a few commonly used interfaces of the Java API.

Interface	Description
`Comparable`	Java contains several comparison operators (e.g., <, <=, >, >=, ==, `!=`) that allow you to compare primitive values. However, these operators *cannot* be used to compare objects. Interface `Comparable` is used to allow objects of a class that `implements` the interface to be compared to one another. Interface `Comparable` is commonly used for ordering objects in a collection such as an array.
`Serializable`	An interface used to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network.
`Runnable`	Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Appendix J). The interface contains one method, `run`, which describes the behavior of an object when executed.
GUI event-listener interfaces	You work with graphical user interfaces (GUIs) every day. In your web browser, you might type the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler.
`SwingConstants`	Contains a set of constants used in GUI programming to position GUI elements on the screen.

Fig. G.28 | Common interfaces of the Java API.

G.14 Wrap-Up

We introduced inheritance—the ability to create classes by absorbing an existing class's members and embellishing them with new capabilities. You learned the notions of superclasses and subclasses and used keyword `extends` to create a subclass that inherits members from a superclass. We showed how to use the `@Override` annotation to prevent unintended overloading by indicating that a method overrides a superclass method. We introduced the access modifier `protected`; subclass methods can directly access `protected` superclass members. You learned how to use `super` to access overridden superclass members. You also saw how constructors are used in inheritance hierarchies. Next, you learned about the methods of class `Object`, the direct or indirect superclass of all Java classes.

We discussed polymorphism—the ability to process objects that share the same superclass in a class hierarchy as if they're all objects of the superclass. We considered how polymorphism makes systems extensible and maintainable, then demonstrated how to use overridden methods to effect polymorphic behavior. We introduced abstract classes, which allow you to provide an appropriate superclass from which other classes can inherit. You learned that an abstract class can declare abstract methods that each subclass must implement to become a concrete class and that a program can use variables of an abstract class to invoke the subclasses' implementations of abstract methods polymorphically. You also learned how to determine an object's type at execution time. We discussed the concepts of `final` methods and classes. Finally, we discussed declaring and implementing an interface as another way to achieve polymorphic behavior.

You should now be familiar with classes, objects, encapsulation, inheritance, polymorphism and interfaces—the most essential aspects of object-oriented programming.

Next, you'll learn about exceptions, useful for handling errors during a program's execution. Exception handling helps you build more robust programs.

Self-Review Exercises (Sections G.1–G.5)

G.1 Fill in the blanks in each of the following statements:

a) _______ is a form of software reusability in which new classes acquire the members of existing classes and embellish those classes with new capabilities.
b) A superclass's _______ members can be accessed in the superclass declaration *and in* subclass declarations.
c) In a(n) _______ relationship, an object of a subclass can also be treated as an object of its superclass.
d) In a(n) _______ relationship, a class object has references to objects of other classes as members.
e) In single inheritance, a class exists in a(n) _______ relationship with its subclasses.
f) A superclass's _______ members are accessible anywhere that the program has a reference to an object of that superclass or to an object of one of its subclasses.
g) When an object of a subclass is instantiated, a superclass _______ is called implicitly or explicitly.
h) Subclass constructors can call superclass constructors via the _______ keyword.

G.2 State whether each of the following is *true* or *false*. If a statement is *false*, explain why.

a) Superclass constructors are not inherited by subclasses.
b) A *has-a* relationship is implemented via inheritance.
c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.
d) When a subclass redefines a superclass method by using the same signature, the subclass is said to overload that superclass method.

Self-Review Exercises (Sections G.6–G.13)

G.3 Fill in the blanks in each of the following statements:

a) If a class contains at least one abstract method, it's a(n) _______ class.
b) Classes from which objects can be instantiated are called _______ classes.
c) _______ involves using a superclass variable to invoke methods on superclass and subclass objects, enabling you to "program in the general."
d) Methods that are not interface methods and that do not provide implementations must be declared using keyword _______.
e) Casting a reference stored in a superclass variable to a subclass type is called _______.

G.4 State whether each of the statements that follows is *true* or *false*. If *false*, explain why.

a) All methods in an `abstract` class must be declared as `abstract` methods.
b) Invoking a subclass-only method through a subclass variable is not allowed.
c) If a superclass declares an `abstract` method, a subclass must implement that method.

d) An object of a class that implements an interface may be thought of as an object of that interface type.

Answers to Self-Review Exercises (Sections G.1–G.5)

G.1 a) Inheritance. b) `public` and `protected`. c) *is-a* or inheritance. d) *has-a* or composition. e) hierarchical. f) `public`. g) constructor. h) `super`.

G.2 a) True. b) False. A *has-a* relationship is implemented via composition. An *is-a* relationship is implemented via inheritance. c) False. This is an example of a *has-a* relationship. Class `Car` has an *is-a* relationship with class `Vehicle`. d) False. This is known as overriding, not overloading—an overloaded method has the same name, but a different signature.

Answers to Self-Review Exercises (Sections G.6–G.13)

G.3 a) abstract. b) concrete. c) Polymorphism. d) `abstract`. e) downcasting.

G.4 a) False. An abstract class can include methods with implementations and `abstract` methods. b) False. Trying to invoke a subclass-only method with a superclass variable is not allowed. c) False. Only a concrete subclass must implement the method. d) True.

Exercises (Sections G.1–G.5)

G.5 What is an `Object` class? Explain its `clone`, `equals` and `finalize` methods along with each of their significance.

G.6 Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. G.2. Use `Student` as the superclass of the hierarchy, then extend `Student` with classes `UndergraduateStudent` and `GraduateStudent`. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, `Freshman`, `Sophomore`, `Junior` and `Senior` might extend `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` might be subclasses of `GraduateStudent`. After drawing the hierarchy, discuss the relationships that exist between the classes. [*Note:* You do not need to write any code for this exercise.]

G.7 Some programmers prefer not to use `protected` access, because they believe it breaks the encapsulation of the superclass. Discuss the relative merits of using `protected` access vs. using `private` access in superclasses.

G.8 Write an inheritance hierarchy for classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` and `Square`. Use `Quadrilateral` as the superclass of the hierarchy. Create and use a `Point` class to represent the points in each shape. Make the hierarchy as deep (i.e., as many levels) as possible. Specify the instance variables and methods for each class. The `private` instance variables of `Quadrilateral` should be the *x-y* coordinate pairs for the four endpoints of the `Quadrilateral`. Write a program that instantiates objects of your classes and outputs each object's area (except `Quadrilateral`).

Exercises (Sections G.6–G.13)

G.9 How does polymorphism enable you to program "in the general" rather than "in the specific"? Discuss the key advantages of programming "in the general."

G.10 What are abstract methods? Describe the circumstances in which an abstract method would be appropriate.

G.11 Describe the relationship between superclasses and subclasses.

G.12 Discuss four ways in which you can assign superclass and subclass references to variables of superclass and subclass types.

G.13 Compare and contrast abstract classes and interfaces. Why would you use an abstract class? Why would you use an interface?

G.14 *(Payroll System Modification)* Modify the payroll system of Figs. G.16–G.21 to include `private` instance variable `birthDate` in class `Employee`. Use class `Date` of Fig. F.7 to represent an employee's birthday. Add *get* methods to class `Date`. Assume that payroll is processed once per month. Create an array of `Employee` variables to store references to the various employee objects. In a loop, calculate the payroll for each `Employee` (polymorphically), and add a $100.00 bonus to the person's payroll amount if the current month is the one in which the `Employee`'s birthday occurs.

G.15 *(Payroll System Modification)* Modify the payroll system of Figs. G.16–G.21 to include an additional `Employee` subclass `PieceWorker` that represents an employee whose pay is based on the number of pieces of merchandise produced. Class `PieceWorker` should contain `private` instance variables `wage` (to store the employee's wage per piece) and `pieces` (to store the number of pieces produced). Provide a concrete implementation of method `earnings` in class `PieceWorker` that calculates the employee's earnings by multiplying the number of pieces produced by the wage per piece. Create an array of `Employee` variables to store references to objects of each concrete class in the new `Employee` hierarchy. For each `Employee`, display its `String` representation and earnings.

G.16 *(Accounts Payable System Modification)* In this exercise, we modify the accounts payable application of Figs. G.23–G.27 to include the complete functionality of the payroll application of Figs. G.16–G.21. The application should still process two `Invoice` objects, but now should process one object of each of the four `Employee` subclasses. If the object currently being processed is a `BasePlusCommissionEmployee`, the application should increase the `BasePlusCommissionEmployee`'s base salary by 10%. Finally, the application should output the payment amount for each object. Complete the following steps to create the new application:

a) Modify classes `HourlyEmployee` (Fig. G.18) and `CommissionEmployee` (Fig. G.19) to place them in the `Payable` hierarchy as subclasses of the version of `Employee` (Fig. G.25) that implements `Payable`. [*Hint:* Change the name of method `earnings` to `getPaymentAmount` in each subclass so that the class satisfies its inherited contract with interface `Payable`.]

b) Modify class `BasePlusCommissionEmployee` (Fig. G.20) such that it extends the version of class `CommissionEmployee` created in part (a).

c) Modify `PayableInterfaceTest` (Fig. G.27) to polymorphically process two `Invoices`, one `SalariedEmployee`, one `HourlyEmployee`, one `CommissionEmployee` and one `BasePlusCommissionEmployee`. First output a `String` representation of each `Payable` object. Next, if an object is a `BasePlusCommissionEmployee`, increase its base salary by 10%. Finally, output the payment amount for each `Payable` object.

G.17 *(Accounts Payable System Modification)* It's possible to include the functionality of the payroll application (Figs. G.16–G.21) in the accounts payable application without modifying `Employee` subclasses `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` or `BasePlusCommissionEmplyee`. To do so, you can modify class `Employee` (Fig. G.16) to implement interface `Payable` and declare method `getPaymentAmount` to invoke method `earnings`. Method `getPaymentAmount` would then be inherited by the subclasses in the `Employee` hierarchy. When `getPaymentAmount` is called for a particular subclass object, it polymorphically invokes the appropriate `earnings` method for that subclass. Reimplement Exercise G.16 using the original `Employee` hierarchy from the payroll application of Figs. G.16–G.21. Modify class `Employee` as described in this exercise, and *do not* modify any of class `Employee`'s subclasses.

Exception Handling: A Deeper Look

H

Objectives

In this appendix you'll:

- Learn what exceptions are and how they're handled.
- Understand when to use exception handling.
- Use `try` blocks to delimit code in which exceptions might occur.
- `throw` exceptions to indicate a problem.
- Use `catch` blocks to specify exception handlers.
- Use the `finally` block to release resources.
- Become familiar with the exception class hierarchy.

Outline

H.1 Introduction

An `exception` is an indication of a problem that occurs during a program's execution. Exception handling enables you to create applications that can resolve (or handle) exceptions. In many cases, handling an exception allows a program to continute executing as if no problem had been encountered. The features presented in this appendix help you write robust programs that can deal with problems and continue executing or terminate gracefully.

H.2 Example: Divide by Zero without Exception Handling

First we demonstrate what happens when errors arise in an application that does not use exception handling. Figure H.1 prompts the user for two integers and passes them to method `quotient`, which calculates the integer quotient and returns an `int` result. In this example, you'll see that exceptions are **thrown** (i.e., the exception occurs) when a method detects a problem and is unable to handle it.

```java
// Fig. H.1: DivideByZeroNoExceptionHandling.java
// Integer division without exception handling.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String[] args )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input

      System.out.print( "Please enter an integer numerator: " );
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer denominator: " );
      int denominator = scanner.nextInt();

```

Fig. H.1 | Integer division without exception handling. (Part 1 of 2.)

```
         int result = quotient( numerator, denominator );
         System.out.printf(
            "\nResult: %d / %d = %d\n", numerator, denominator, result );
      } // end main
   } // end class DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
            DivideByZeroNoExceptionHandling.java:20)
```

Fig. H.1 | Integer division without exception handling. (Part 2 of 2.)

The first sample execution in Fig. H.1 shows a successful division. In the second execution, the user enters the value `0` as the denominator. Several lines of information are displayed in response to this invalid input. This information is known as a **stack trace**, which includes the name of the exception (`java.lang.ArithmeticException`) in a descriptive message that indicates the problem that occurred and the method-call stack (i.e., the call chain) at the time it occurred. The stack trace includes the path of execution that led to the exception method by method. This helps you debug the program. The first line specifies that an `ArithmeticException` has occurred. The text after the name of the exception ("`/ by zero`") indicates that this exception occurred as a result of an attempt to divide by zero. Java does not allow division by zero in integer arithmetic. When this occurs, Java throws an **`ArithmeticException`**. `ArithmeticException`s can arise from a number of different problems in arithmetic, so the extra data ("`/ by zero`") provides more specific information. Java *does* allow division by zero with floating-point values. Such a calculation results in the value positive or negative infinity, which is represented in Java as a floating-point value (but displays as the string `Infinity` or `-Infinity`). If 0.0 is divided by 0.0, the result is NaN (not a number), which is also represented in Java as a floating-point value (but displays as `NaN`).

Starting from the last line of the stack trace, we see that the exception was detected in line 22 of method `main`. Each line of the stack trace contains the class name and method (`DivideByZeroNoExceptionHandling.main`) followed by the file name and line number (`DivideByZeroNoExceptionHandling.java:22`). Moving up the stack trace, we see that the exception occurs in line 10, in method `quotient`. The top row of the call chain indicates the **throw point**—the initial point at which the exception occurs. The throw point of this exception is in line 10 of method `quotient`.

In the third execution, the user enters the string `"hello"` as the denominator. Notice again that a stack trace is displayed. This informs us that an `InputMismatchException` has occurred (package `java.util`). Our prior examples that read numeric values from the user assumed that the user would input a proper integer value. However, users sometimes make mistakes and input noninteger values. An **`InputMismatchException`** occurs when `Scanner` method `nextInt` receives a `string` that does not represent a valid integer. Starting from the end of the stack trace, we see that the exception was detected in line 20 of method `main`. Moving up the stack trace, we see that the exception occurred in method `nextInt`. Notice that in place of the file name and line number, we're provided with the text `Unknown Source`. This means that the so-called debugging symbols that provide the filename and line number information for that method's class were not available to the JVM—this is typically the case for the classes of the Java API. Many IDEs have access to the Java API source code and will display file names and line numbers in stack traces.

In the sample executions of Fig. H.1 when exceptions occur and stack traces are displayed, the program also exits. This does not always occur in Java—sometimes a program may continue even though an exception has occurred and a stack trace has been printed. In such cases, the application may produce unexpected results. For example, a graphical user interface (GUI) application will often continue executing. The next section demonstrates how to handle these exceptions.

In Fig. H.1 both types of exceptions were detected in method `main`. In the next example, we'll see how to handle these exceptions to enable the program to run to normal completion.

H.3 Example: Handling `ArithmeticExceptions` and `InputMismatchExceptions`

The application in Fig. H.2, which is based on Fig. H.1, uses exception handling to process any `ArithmeticExceptions` and `InputMistmatchExceptions` that arise. The application still prompts the user for two integers and passes them to method `quotient`, which calculates the quotient and returns an `int` result. This version of the application uses exception handling so that if the user makes a mistake, the program catches and handles (i.e., deals with) the exception—in this case, allowing the user to enter the input again.

```java
1  // Fig. H.2: DivideByZeroWithExceptionHandling.java
2  // Handling ArithmeticExceptions and InputMismatchExceptions.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
```

Fig. H.2 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 1 of 3.)

```java

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numerator, int denominator )
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String[] args )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean continueLoop = true; // determines if more input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominator: " );
            int denominator = scanner.nextInt();

            int result = quotient( numerator, denominator );
            System.out.printf( "\nResult: %d / %d = %d\n", numerator,
               denominator, result );
            continueLoop = false; // input successful; end looping
         } // end try
         catch ( InputMismatchException inputMismatchException )
         {
            System.err.printf( "\nException: %s\n",
               inputMismatchException );
            scanner.nextLine(); // discard input so user can try again
            System.out.println(
               "You must enter integers. Please try again.\n" );
         } // end catch
         catch ( ArithmeticException arithmeticException )
         {
            System.err.printf( "\nException: %s\n", arithmeticException );
            System.out.println(
               "Zero is an invalid denominator. Please try again.\n" );
         } // end catch
      } while ( continueLoop ); // end do...while
   } // end main
} // end class DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. H.2 | Handling ArithmeticExceptions and InputMismatchExceptions. (Part 2 of 3.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. H.2 | Handling `ArithmeticExceptions` and `InputMismatchExceptions`. (Part 3 of 3.)

The first sample execution in Fig. H.2 is a successful one that does not encounter any problems. In the second execution the user enters a zero denominator, and an `ArithmeticException` exception occurs. In the third execution the user enters the string `"hello"` as the denominator, and an `InputMismatchException` occurs. For each exception, the user is informed of the mistake and asked to try again, then is prompted for two new integers. In each sample execution, the program runs successfully to completion.

Class `InputMismatchException` is imported in line 3. Class `ArithmeticException` does not need to be imported because it's in package `java.lang`. Line 18 creates the `boolean` variable `continueLoop`, which is `true` if the user has not yet entered valid input. Lines 20–48 repeatedly ask users for input until a valid input is received.

Enclosing Code in a `try` Block

Lines 22–33 contain a **`try` block**, which encloses the code that might `throw` an exception and the code that should not execute if an exception occurs (i.e., if an exception occurs, the remaining code in the `try` block will be skipped). A `try` block consists of the keyword `try` followed by a block of code enclosed in curly braces. [*Note:* The term "`try` block" sometimes refers only to the block of code that follows the `try` keyword (not including the `try` keyword itself). For simplicity, we use the term "`try` block" to refer to the block of code that follows the `try` keyword, as well as the `try` keyword.] The statements that read the integers from the keyboard (lines 25 and 27) each use method `nextInt` to read an `int` value. Method `nextInt` throws an `InputMismatchException` if the value read in is not an integer.

The division that can cause an `ArithmeticException` is not performed in the `try` block. Rather, the call to method `quotient` (line 29) invokes the code that attempts the division (line 12); the JVM throws an `ArithmeticException` object when the denominator is zero.

Software Engineering Observation H.1

Exceptions may surface through explicitly mentioned code in a `try` *block, through calls to other methods, through deeply nested method calls initiated by code in a* `try` *block or from the Java Virtual Machine as it executes Java bytecodes.*

Catching Exceptions

The `try` block in this example is followed by two `catch` blocks—one that handles an `InputMismatchException` (lines 34–41) and one that handles an `ArithmeticException` (lines 42–47). A **`catch` block** (also called a **`catch` clause** or **exception handler**) catches (i.e., receives) and handles an exception. A `catch` block begins with the keyword `catch` and is followed by a parameter in parentheses (called the exception parameter, discussed shortly) and a block of code enclosed in curly braces. [*Note:* The term "`catch` clause" is sometimes used to refer to the keyword `catch` followed by a block of code, whereas the term "`catch` block" refers to only the block of code following the `catch` keyword, but not including it. For simplicity, we use the term "`catch` block" to refer to the block of code following the `catch` keyword, as well as the keyword itself.]

At least one `catch` block or a **`finally` block** (discussed in Section H.6) must immediately follow the `try` block. Each `catch` block specifies in parentheses an **exception parameter** that identifies the exception type the handler can process. When an exception occurs in a `try` block, the `catch` block that executes is the *first* one whose type matches the type of the exception that occurred (i.e., the type in the `catch` block matches the thrown exception type exactly or is a superclass of it). The exception parameter's name enables the `catch` block to interact with a caught exception object—e.g., to implicitly invoke the caught exception's `toString` method (as in lines 37 and 44), which displays basic information about the exception. Notice that we use the **`System.err` (standard error stream) object** to output error messages. By default, `System.err`'s print methods, like those of `System.out`, display data to the command prompt.

Line 38 of the first `catch` block calls `Scanner` method `nextLine`. Because an `InputMismatchException` occurred, the call to method `nextInt` never successfully read in the user's data—so we read that input with a call to method `nextLine`. We do not do anything with the input at this point, because we know that it's invalid. Each `catch` block displays an error message and asks the user to try again. After either `catch` block terminates, the user is prompted for input. We'll soon take a deeper look at how this flow of control works in exception handling.

Common Programming Error H.1

It's a syntax error to place code between a `try` *block and its corresponding* `catch` *blocks.*

Common Programming Error H.2

Each `catch` *block can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.*

An **uncaught exception** is one for which there are no matching `catch` blocks. You saw uncaught exceptions in the second and third outputs of Fig. H.1. Recall that when exceptions occurred in that example, the application terminated early (after displaying the exception's stack trace). This does not always occur as a result of uncaught exceptions. Java uses a "multithreaded" model of program execution—each **thread** is a parallel activity. One program can have many threads. If a program has only one thread, an uncaught exception will cause the program to terminate. If a program has multiple threads, an uncaught exception will terminate *only* the thread where the exception occurred. In such programs, however, certain threads may rely on others, and if one thread terminates due to an uncaught exception, there may be adverse effects to the rest of the program. Appendix J discusses these issues.

Termination Model of Exception Handling

If an exception occurs in a `try` block (such as an `InputMismatchException` being thrown as a result of the code at line 25 of Fig. H.2), the `try` block terminates immediately and program control transfers to the *first* of the following `catch` blocks in which the exception parameter's type matches the thrown exception's type. In Fig. H.2, the first `catch` block catches `InputMismatchExceptions` (which occur if invalid input is entered) and the second `catch` block catches `ArithmeticExceptions` (which occur if an attempt is made to divide by zero). After the exception is handled, program control does *not* return to the throw point, because the `try` block has *expired* (and its local variables have been lost). Rather, control resumes after the last `catch` block. This is known as the **termination model of exception handling**. Some languages use the **resumption model of exception handling**, in which, after an exception is handled, control resumes just after the throw point.

Notice that we name our exception parameters (`inputMismatchException` and `arithmeticException`) based on their type. Java programmers often simply use the letter `e` as the name of their exception parameters.

After executing a `catch` block, this program's flow of control proceeds to the first statement after the last `catch` block (line 48 in this case). The condition in the `do...while` statement is `true` (variable `continueLoop` contains its initial value of `true`), so control returns to the beginning of the loop and the user is once again prompted for input. This control statement will loop until valid input is entered. At that point, program control reaches line 32, which assigns `false` to variable `continueLoop`. The `try` block then terminates. If no exceptions are thrown in the `try` block, the `catch` blocks are skipped and control continues with the first statement after the `catch` blocks (we'll learn about another possibility when we discuss the `finally` block in Section H.6). Now the condition for the `do...while` loop is `false`, and method `main` ends.

The `try` block and its corresponding `catch` and/or `finally` blocks form a **`try` statement**. Do not confuse the terms "`try` block" and "`try` statement"—the latter includes the `try` block as well as the following `catch` blocks and/or `finally` block.

As with any other block of code, when a `try` block terminates, local variables declared in the block go out of scope and are no longer accessible; thus, the local variables of a `try` block are not accessible in the corresponding `catch` blocks. When a `catch` block terminates, local variables declared within the `catch` block (including the exception parameter of that `catch` block) also go out of scope and are destroyed. Any remaining `catch` blocks in the `try` statement are ignored, and execution resumes at the first line of code after the `try...catch` sequence—this will be a `finally` block, if one is present.

Using the `throws` *Clause*

Now let's examine method `quotient` (Fig. H.2, lines 9–13). The portion of the method declaration located at line 10 is known as a **`throws` clause**. It specifies the exceptions the method throws. This clause appears *after* the method's parameter list and *before* the method's body. It contains a comma-separated list of the exceptions that the method will throw if various problems occur. Such exceptions may be thrown by statements in the method's body or by methods called from the body. A method can throw exceptions of the classes listed in its `throws` clause or of their subclasses. We've added the `throws` clause to this application to indicate to the rest of the program that this method may throw an `ArithmeticException`. Clients of method `quotient` are thus informed that the method may throw an `ArithmeticException`. You'll learn more about the `throws` clause in Section H.5.

When line 12 executes, if the `denominator` is zero, the JVM throws an `ArithmeticException` object. This object will be caught by the `catch` block at lines 42–47, which displays basic information about the exception by implicitly invoking the exception's `toString` method, then asks the user to try again.

If the `denominator` is not zero, method `quotient` performs the division and returns the result to the point of invocation of method `quotient` in the `try` block (line 29). Lines 30–31 display the result of the calculation and line 32 sets `continueLoop` to `false`. In this case, the `try` block completes successfully, so the program skips the `catch` blocks and fails the condition at line 48, and method `main` completes execution normally.

When `quotient` throws an `ArithmeticException`, `quotient` terminates and does not return a value, and `quotient`'s local variables go out of scope (and are destroyed). If `quotient` contained local variables that were references to objects and there were no other references to those objects, the objects would be marked for garbage collection. Also, when an exception occurs, the `try` block from which `quotient` was called terminates before lines 30–32 can execute. Here, too, if local variables were created in the `try` block prior to the exception's being thrown, these variables would go out of scope.

If an `InputMismatchException` is generated by lines 25 or 27, the `try` block terminates and execution continues with the `catch` block at lines 34–41. In this case, method `quotient` is not called. Then method `main` continues after the last `catch` block (line 48).

H.4 When to Use Exception Handling

Exception handling is designed to process **synchronous errors**, which occur when a statement executes. Common examples we'll see throughout the book are out-of-range array indices, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid method parameters, thread interruption (as we'll see in Appendix J) and unsuccessful memory allocation (due to lack of memory). Exception handling is not designed to process problems associated with **asynchronous events** (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

H.5 Java Exception Hierarchy

All Java exception classes inherit directly or indirectly from class **`Exception`**, forming an inheritance hierarchy. You can extend this hierarchy with your own exception classes. Class **`Throwable`** (a subclass of `Object`) is the superclass of class `Exception`. Only `Throwable` objects can be used with the exception-handling mechanism. Class `Throwable` has two subclasses: `Exception` and `Error`. Class `Exception` and its subclasses—for instance, `RuntimeException` (package `java.lang`) and `IOException` (package `java.io`)—represent exceptional situations that can occur in a Java program and that can be caught by the application. Class **Error** and its subclasses represent abnormal situations that happen in the JVM. Most *Errors happen infrequently and should not be caught by applications—it's usually not possible for applications to recover from Errors.*

Checked vs. Unchecked Exceptions

Java distinguishes between **checked exceptions** and **unchecked exceptions**. This distinction is important, because the Java compiler enforces a **catch-or-declare requirement** for checked exceptions. An exception's type determines whether it's checked or unchecked. All exception types that are direct or indirect subclasses of class **RuntimeException** (package java.lang) are unchecked exceptions. These are typically caused by defects in your program's code. Examples of unchecked exceptions include ArrayIndexOutOfBoundsExceptions (discussed in Appendix E) and ArithmeticExceptions. All classes that inherit from class Exception but not class RuntimeException are considered to be checked exceptions. Such exceptions are typically caused by conditions that are not under the control of the program—for example, in file processing, the program can't open a file because the file does not exist. Classes that inherit from class Error are considered to be unchecked.

The compiler *checks* each method call and method declaration to determine whether the method throws checked exceptions. If so, the compiler verifies that the checked exception is caught or is declared in a throws clause. We show how to catch and declare checked exceptions in the next several examples. Recall from Section H.3 that the throws clause specifies the exceptions a method throws. Such exceptions are not caught in the method's body. To satisfy the *catch* part of the catch-or-declare requirement, the code that generates the exception must be wrapped in a try block and must provide a catch handler for the checked-exception type (or one of its superclass types). To satisfy the *declare* part of the catch-or-declare requirement, the method containing the code that generates the exception must provide a throws clause containing the checked-exception type after its parameter list and before its method body. If the catch-or-declare requirement is not satisfied, the compiler will issue an error message indicating that the exception must be caught or declared. This forces you to think about the problems that may occur when a method that throws checked exceptions is called.

Software Engineering Observation H.2

You must deal with checked exceptions. This results in more robust code than would be created if you were able to simply ignore the exceptions.

Common Programming Error H.3

A compilation error occurs if a method explicitly attempts to throw a checked exception (or calls another method that throws a checked exception) and that exception is not listed in that method's throws clause.

Common Programming Error H.4

If a subclass method overrides a superclass method, it's an error for the subclass method to list more exceptions in its throws clause than the overridden superclass method does. However, a subclass's throws clause can contain a subset of a superclass's throws list.

Software Engineering Observation H.3

If your method calls other methods that throw checked exceptions, those exceptions must be caught or declared in your method. If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.

Unlike checked exceptions, the Java compiler does *not* check the code to determine whether an unchecked exception is caught or declared. Unchecked exceptions typically can be prevented by proper coding. For example, the unchecked `ArithmeticException` thrown by method `quotient` (lines 9–13) in Fig. H.2 can be avoided if the method ensures that the denominator is not zero *before* attempting to perform the division. Unchecked exceptions are not required to be listed in a method's `throws` clause—even if they are, it's not required that such exceptions be caught by an application.

Software Engineering Observation H.4

Although the compiler does not enforce the catch-or-declare requirement for unchecked exceptions, provide appropriate exception-handling code when it's known that such exceptions might occur. For example, a program should process the `NumberFormatException` *from* `Integer` *method* `parseInt`*, even though* `NumberFormatException` *(an indirect subclass of* `RuntimeException`*) is an unchecked exception type. This makes your programs more robust.*

Catching Subclass Exceptions

If a `catch` handler is written to catch superclass-type exception objects, it can also catch all objects of that class's subclasses. This enables `catch` to handle related errors with a concise notation and allows for polymorphic processing of related exceptions. You can certainly catch each subclass type individually if those exceptions require different processing.

Only the First Matching `catch` *Executes*

If there are *multiple* `catch` blocks that match a particular exception type, only the *first* matching `catch` block executes when an exception of that type occurs. It's a compilation error to catch the *exact same type* in two different `catch` blocks associated with a particular `try` block. However, there may be several `catch` blocks that match an exception—i.e., several `catch` blocks whose types are the same as the exception type or a superclass of that type. For instance, we could follow a `catch` block for type `ArithmeticException` with a `catch` block for type `Exception`—both would match `ArithmeticExceptions`, but only the first matching `catch` block would execute.

Error-Prevention Tip H.1

Catching subclass types individually is subject to error if you forget to test for one or more of the subclass types explicitly; catching the superclass guarantees that objects of all subclasses will be caught. Positioning a `catch` *block for the superclass type after all other subclass* `catch` *blocks ensures that all subclass exceptions are eventually caught.*

Common Programming Error H.5

Placing a `catch` *block for a superclass exception type before other* `catch` *blocks that catch subclass exception types would prevent those* `catch` *blocks from executing, so a compilation error occurs.*

H.6 `finally` Block

Programs that obtain certain types of resources must return them to the system explicitly to avoid so-called **resource leaks.** In programming languages such as C and C++, the most common kind of resource leak is a memory leak. Java performs automatic garbage collection of memory no longer used by programs, thus avoiding most memory leaks. However, other types of resource leaks can occur. For example, files, database connections and network connections that are not closed properly after they're no longer needed might not be available for use in other programs.

Error-Prevention Tip H.2

A subtle issue is that Java does not entirely eliminate memory leaks. Java will not garbage-collect an object until there are no remaining references to it. Thus, if you erroneously keep references to unwanted objects, memory leaks can occur. To help avoid this problem, set reference-type variables to `null` when they're no longer needed.

The `finally` block (which consists of the `finally` keyword, followed by code enclosed in curly braces), sometimes referred to as the **`finally` clause**, is optional. If it's present, it's placed after the last `catch` block. If there are no `catch` blocks, the `finally` block immediately follows the `try` block.

The `finally` block will execute whether or not an exception is thrown in the corresponding `try` block. The `finally` block also will execute if a `try` block exits by using a `return`, `break` or `continue` statement or simply by reaching its closing right brace. The `finally` block will *not* execute if the application exits early from a `try` block by calling method **`System.exit`**. This method immediately terminates an application.

Because a `finally` block almost always executes, it typically contains resource-release code. Suppose a resource is allocated in a `try` block. If no exception occurs, the `catch` blocks are skipped and control proceeds to the `finally` block, which frees the resource. Control then proceeds to the first statement after the `finally` block. If an exception occurs in the `try` block, the `try` block terminates. If the program catches the exception in one of the corresponding `catch` blocks, it processes the exception, then the `finally` block releases the resource and control proceeds to the first statement after the `finally` block. If the program doesn't catch the exception, the `finally` block *still* releases the resource and an attempt is made to catch the exception in a calling method.

Error-Prevention Tip H.3

The `finally` block is an ideal place to release resources acquired in a `try` block (such as opened files), which helps eliminate resource leaks.

Performance Tip H.1

Always release a resource explicitly and at the earliest possible moment at which it's no longer needed. This makes resources available for reuse as early as possible, thus improving resource utilization.

If an exception that occurs in a `try` block cannot be caught by one of that `try` block's `catch` handlers, the program skips the rest of the `try` block and control proceeds to the `finally` block. Then the program passes the exception to the next outer `try` block—normally in the calling method—where an associated catch block might catch it. This process can occur through many levels of try blocks. Also, the exception could go uncaught.

If a catch block throws an exception, the finally block still executes. Then the exception is passed to the next outer try block—again, normally in the calling method.

Figure H.3 demonstrates that the finally block executes even if an exception is not thrown in the corresponding try block. The program contains static methods main (lines 6–18), throwException (lines 21–44) and doesNotThrowException (lines 47–64). Methods throwException and doesNotThrowException are declared static, so main can call them directly without instantiating a UsingExceptions object.

```
 1  // Fig. H.3: UsingExceptions.java
 2  // try...catch...finally exception handling mechanism.
 3
 4  public class UsingExceptions
 5  {
 6     public static void main( String[] args )
 7     {
 8        try
 9        {
10           throwException(); // call method throwException
11        } // end try
12        catch ( Exception exception ) // exception thrown by throwException
13        {
14           System.err.println( "Exception handled in main" );
15        } // end catch
16
17        doesNotThrowException();
18     } // end main
19
20     // demonstrate try...catch...finally
21     public static void throwException() throws Exception
22     {
23        try // throw an exception and immediately catch it
24        {
25           System.out.println( "Method throwException" );
26           throw new Exception(); // generate exception
27        } // end try
28        catch ( Exception exception ) // catch exception thrown in try
29        {
30           System.err.println(
31              "Exception handled in method throwException" );
32           throw exception; // rethrow for further processing
33
34           // code here would not be reached; would cause compilation errors
35
36        } // end catch
37        finally // executes regardless of what occurs in try...catch
38        {
39           System.err.println( "Finally executed in throwException" );
40        } // end finally
41
```

Fig. H.3 | try...catch...finally exception-handling mechanism. (Part 1 of 2.)

```
42          // code here would not be reached; would cause compilation errors
43
44       } // end method throwException
45
46       // demonstrate finally when no exception occurs
47       public static void doesNotThrowException()
48       {
49          try // try block does not throw an exception
50          {
51             System.out.println( "Method doesNotThrowException" );
52          } // end try
53          catch ( Exception exception ) // does not execute
54          {
55             System.err.println( exception );
56          } // end catch
57          finally // executes regardless of what occurs in try...catch
58          {
59             System.err.println(
60                "Finally executed in doesNotThrowException" );
61          } // end finally
62
63          System.out.println( "End of method doesNotThrowException" );
64       } // end method doesNotThrowException
65    } // end class UsingExceptions
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Fig. H.3 | `try`...`catch`...`finally` exception-handling mechanism. (Part 2 of 2.)

`System.out` and `System.err` are **streams**—sequences of bytes. While `System.out` (known as the **standard output stream**) displays a program's output, `System.err` (known as the **standard error stream**) displays a program's errors. Output from these streams can be redirected (i.e., sent to somewhere other than the command prompt, such as to a file). Using two different streams enables you to easily separate error messages from other output. For instance, data output from `System.err` could be sent to a log file, while data output from `System.out` can be displayed on the screen. For simplicity, this appendix will not redirect output from `System.err`, but will display such messages to the command prompt. You'll learn more about streams in Appendix J.

Throwing Exceptions Using the `throw` *Statement*

Method `main` (Fig. H.3) begins executing, enters its `try` block and immediately calls method `throwException` (line 10). Method `throwException` throws an `Exception`. The statement at line 26 is known as a **`throw` statement**—it's executed to indicate that an exception has occurred. So far, you've only caught exceptions thrown by called methods.

You can throw exceptions yourself by using the `throw` statement. Just as with exceptions thrown by the Java API's methods, this indicates to client applications that an error has occurred. A `throw` statement specifies an object to be thrown. The operand of a `throw` can be of any class derived from class `Throwable`.

Software Engineering Observation H.5

When `toString` is invoked on any `Throwable` object, its resulting `string` includes the descriptive `string` that was supplied to the constructor, or simply the class name if no `string` was supplied.

Software Engineering Observation H.6

An object can be thrown without containing information about the problem that occurred. In this case, simply knowing that an exception of a particular type occurred may provide sufficient information for the handler to process the problem correctly.

Software Engineering Observation H.7

Exceptions can be thrown from constructors. When an error is detected in a constructor, an exception should be thrown to avoid creating an improperly formed object.

Rethrowing Exceptions

Line 32 of Fig. H.3 **rethrows the exception**. Exceptions are rethrown when a `catch` block, upon receiving an exception, decides either that it cannot process that exception or that it can only partially process it. Rethrowing an exception defers the exception handling (or perhaps a portion of it) to another `catch` block associated with an outer `try` statement. An exception is rethrown by using the **`throw` keyword**, followed by a reference to the exception object that was just caught. Exceptions cannot be rethrown from a `finally` block, as the exception parameter (a local variable) from the `catch` block no longer exists.

When a rethrow occurs, the *next enclosing `try` block* detects the rethrown exception, and that `try` block's `catch` blocks attempt to handle it. In this case, the next enclosing `try` block is found at lines 8–11 in method `main`. Before the rethrown exception is handled, however, the `finally` block (lines 37–40) executes. Then method `main` detects the rethrown exception in the `try` block and handles it in the `catch` block (lines 12–15).

Next, `main` calls method `doesNotThrowException` (line 17). No exception is thrown in `doesNotThrowException`'s `try` block (lines 49–52), so the program skips the `catch` block (lines 53–56), but the `finally` block (lines 57–61) nevertheless executes. Control proceeds to the statement after the `finally` block (line 63). Then control returns to `main` and the program terminates.

Common Programming Error H.6

If an exception has not been caught when control enters a `finally` block and the `finally` block throws an exception that's not caught in the `finally` block, the first exception will be lost *and the exception from the `finally` block will be returned to the calling method.*

Error-Prevention Tip H.4

Avoid placing code that can `throw` an exception in a `finally` block. If such code is required, enclose the code in a `try...catch` within the `finally` block.

Common Programming Error H.7

Assuming that an exception thrown from a `catch` block will be processed by that `catch` block or any other `catch` block associated with the same `try` statement can lead to logic errors.

Good Programming Practice H.1

Exception handling is intended to remove error-processing code from the main line of a program's code to improve program clarity. Do not place `try...catch... finally` around every statement that may throw an exception. This makes programs difficult to read. Rather, place one `try` block around a significant portion of your code, follow that `try` block with `catch` blocks that handle each possible exception and follow the `catch` blocks with a single `finally` block (if one is required).

H.7 Stack Unwinding and Obtaining Information from an Exception Object

When an exception is thrown but not caught in a particular scope, the method-call stack is "unwound," and an attempt is made to `catch` the exception in the next outer `try` block. This process is called **stack unwinding**. Unwinding the method-call stack means that the method in which the exception was not caught *terminates*, all local variables in that method go out of scope and control returns to the statement that originally invoked that method. If a `try` block encloses that statement, an attempt is made to `catch` the exception. If a `try` block does not enclose that statement or if the exception is not caught, stack unwinding occurs again. Figure H.4 demonstrates stack unwinding, and the exception handler in `main` shows how to access the data in an exception object.

```
// Fig. H.4: UsingExceptions.java
// Stack unwinding and obtaining data from an exception object.

public class UsingExceptions
{
   public static void main( String[] args )
   {
      try
      {
         method1(); // call method1
      } // end try
      catch ( Exception exception ) // catch exception thrown in method1
      {
         System.err.printf( "%s\n\n", exception.getMessage() );
         exception.printStackTrace(); // print exception stack trace

         // obtain the stack-trace information
         StackTraceElement[] traceElements = exception.getStackTrace();

         System.out.println( "\nStack trace from getStackTrace:" );
         System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );

```

Fig. H.4 | Stack unwinding and obtaining data from an exception object. (Part 1 of 2.)

```
23          // loop through traceElements to get exception description
24          for ( StackTraceElement element : traceElements )
25          {
26             System.out.printf( "%s\t", element.getClassName() );
27             System.out.printf( "%s\t", element.getFileName() );
28             System.out.printf( "%s\t", element.getLineNumber() );
29             System.out.printf( "%s\n", element.getMethodName() );
30          } // end for
31       } // end catch
32    } // end main
33
34    // call method2; throw exceptions back to main
35    public static void method1() throws Exception
36    {
37       method2();
38    } // end method method1
39
40    // call method3; throw exceptions back to method1
41    public static void method2() throws Exception
42    {
43       method3();
44    } // end method method2
45
46    // throw Exception back to method2
47    public static void method3() throws Exception
48    {
49       throw new Exception( "Exception thrown in method3" );
50    } // end method method3
51 } // end class UsingExceptions
```

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:49)
        at UsingExceptions.method2(UsingExceptions.java:43)
        at UsingExceptions.method1(UsingExceptions.java:37)
        at UsingExceptions.main(UsingExceptions.java:10)

Stack trace from getStackTrace:
Class           File                    Line    Method
UsingExceptions UsingExceptions.java    49      method3
UsingExceptions UsingExceptions.java    43      method2
UsingExceptions UsingExceptions.java    37      method1
UsingExceptions UsingExceptions.java    10      main
```

Fig. H.4 | Stack unwinding and obtaining data from an exception object. (Part 2 of 2.)

Stack Unwinding

In `main`, the `try` block (lines 8–11) calls `method1` (declared at lines 35–38), which in turn calls `method2` (declared at lines 41–44), which in turn calls `method3` (declared at lines 47–50). Line 49 of `method3` throws an `Exception` object—this is the *throw point*. Because the `throw` statement at line 49 is *not* enclosed in a `try` block, *stack unwinding* occurs—`method3` terminates at line 49, then returns control to the statement in `method2` that invoked `method3` (i.e., line 43). Because *no* `try` block encloses line 43, *stack unwinding* occurs again—`method2` terminates at line 43 and returns control to the statement in `method1` that invoked `method2` (i.e., line 37). Because *no* `try` block encloses line 37, *stack unwinding* occurs one more time—`method1` terminates at line 37 and returns control to the statement in `main` that invoked `method1` (i.e., line 10). The `try` block at lines 8–11 encloses this statement. The exception has not been handled, so the `try` block terminates and the first matching `catch` block (lines 12–31) catches and processes the exception. If there were no matching `catch` blocks, and the exception is not declared in each method that throws it, a compilation error would occur. Remember that this is not always the case—for *unchecked* exceptions, the application will compile, but it will run with unexpected results.

Obtaining Data from an Exception Object

Recall that exceptions derive from class `Throwable`. Class `Throwable` offers a **`printStackTrace`** method that outputs to the standard error stream the stack trace (discussed in Section H.2). Often, this is helpful in testing and debugging. Class `Throwable` also provides a **`getStackTrace`** method that retrieves the stack-trace information that might be printed by `printStackTrace`. Class `Throwable`'s **`getMessage`** method returns the descriptive string stored in an exception.

Error-Prevention Tip H.5

An exception that's not caught in an application causes Java's default exception handler *to run. This displays the name of the exception, a descriptive message that indicates the problem that occurred and a complete execution stack trace. In an application with a single thread of execution, the application terminates. In an application with multiple threads, the thread that caused the exception terminates.*

Error-Prevention Tip H.6

`Throwable` method `toString` (inherited by all `Throwable` subclasses) returns a `String` containing the name of the exception's class and a descriptive message.

The `catch` handler in Fig. H.4 (lines 12–31) demonstrates `getMessage`, `printStackTrace` and `getStackTrace`. If we wanted to output the stack-trace information to streams other than the standard error stream, we could use the information returned from `getStackTrace` and output it to another stream or use one of the overloaded versions of method `printStackTrace`.

Line 14 invokes the exception's `getMessage` method to get the exception description. Line 15 invokes the exception's `printStackTrace` method to output the stack trace that indicates where the exception occurred. Line 18 invokes the exception's `getStackTrace` method to obtain the stack-trace information as an array of **`StackTraceElement`** objects. Lines 24–30 get each `StackTraceElement` in the array and invoke its methods **`getClassName`**, **`getFileName`**, **`getLineNumber`** and **`getMethodName`** to get the class name, file name, line number and method name, respectively, for that `StackTraceElement`. Each `StackTraceElement` represents one method call on the method-call stack.

The program's output shows that the stack-trace information printed by `printStackTrace` follows the pattern: *className*.*methodName*(*fileName*:*lineNumber*), where *className*, *methodName* and *fileName* indicate the names of the class, method and file in which the exception occurred, respectively, and the *lineNumber* indicates where in the file the exception occurred. You saw this in the output for Fig. H.1. Method `getStackTrace` enables custom processing of the exception information. Compare the output of `printStackTrace` with the output created from the `StackTraceElements` to see that both contain the same stack-trace information.

Software Engineering Observation H.8

Never provide a `catch` handler with an empty body—this effectively ignores the exception. At least use `printStackTrace` to output an error message to indicate that a problem exists.

H.8 Wrap-Up

In this appendix, you learned how to use exception handling to deal with errors. You learned that exception handling enables you to remove error-handling code from the "main line" of the program's execution. We showed how to use `try` blocks to enclose code that may throw an exception, and how to use `catch` blocks to deal with exceptions that may arise. You learned about the termination model of exception handling, which dictates that after an exception is handled, program control does not return to the throw point. We discussed checked vs. unchecked exceptions, and how to specify with the `throws` clause the exceptions that a method might throw. You learned how to use the `finally` block to release resources whether or not an exception occurs. You also learned how to throw and rethrow exceptions. We showed how to obtain information about an exception using methods `printStackTrace`, `getStackTrace` and `getMessage`. In the next appendix, we discuss graphical user interface concepts and explain the essentials of event handling.

Self-Review Exercises

H.1 What does the information called stack trace contain?

H.2 Give several reasons why exception-handling techniques should not be used for conventional program control.

H.3 Will the `finally` block be executed only when an exception is thrown? If it is used in a Java program, where should it be placed?

H.4 When does an `InputMismatchException` occur?

H.5 If no exceptions are thrown in a `try` block, where does control proceed to when the `try` block completes execution?

H.6 Give a key advantage of using `catch( Exception` *exceptionName* `)`.

H.7 Is it mandatory to have both the `catch` and `finally` blocks after a `try` block?

H.8 What happens if no `catch` handler matches the type of a thrown object?

H.9 What happens if several `catch` blocks match the type of the thrown object?

H.10 Briefly explain the Java Exception Hierarchy.

H.11 What is the key reason for using `finally` blocks?

H.12 What happens when a `catch` block throws an `Exception`?

H.13 What does the statement `throw` *exceptionReference* do in a `catch` block?

H.14 What happens to a local reference in a `try` block when that block throws an `Exception`?

Answers to Self-Review Exercises

H.1 It contains the name of the exception in a descriptive error message, which indicates the problem that occurred and the method-call stack at the time, along with the path of execution that led to the exception method by method.

H.2 (a) Exception handling is designed to handle infrequently occurring situations that often result in program termination, not situations that arise all the time. (b) Flow of control with conventional control structures is generally clearer and more efficient than with exceptions. (c) The additional exceptions can get in the way of genuine error-type exceptions. It becomes more difficult for you to keep track of the larger number of exception cases.

H.3 No. The `finally` block will execute whether or not an exception is thrown in the `try` block. It should be placed after the last catch block.

H.4 An `InputMismatchException` occurs when a `Scanner` method receives data of a type that is different from what it expects.

H.5 The `catch` blocks for that `try` statement are skipped, and the program resumes execution after the last `catch` block. If there's a `finally` block, it's executed first; then the program resumes execution after the `finally` block.

H.6 The form `catch( Exception` *exceptionName* `)` catches any type of exception thrown in a `try` block. An advantage is that no thrown `Exception` can slip by without being caught. You can then decide to handle the exception or possibly rethrow it.

H.7 No. At least one `catch` block or a `finally` block must immediately follow the `try` block.

H.8 This causes the search for a match to continue in the next enclosing `try` statement. If there's a `finally` block, it will be executed before the exception goes to the next enclosing `try` statement. If there are no enclosing `try` statements for which there are matching `catch` blocks and the exceptions are declared (or unchecked), a stack trace is printed and the current thread terminates early. If the exceptions are checked, but not caught or declared, compilation errors occur.

H.9 The first matching `catch` block after the `try` block is executed.

H.10 All Java exception classes inherit either directly or indirectly from class `Exception`. Using one's own exception classes, this hierarchy can be extended. Class `Throwable` is the superclass of class `Exception`.

H.11 The `finally` block is the preferred means for releasing resources to prevent resource leaks.

H.12 First, control passes to the `finally` block if there is one. Then the exception will be processed by a `catch` block (if one exists) associated with an enclosing `try` block (if one exists).

H.13 It rethrows the exception for processing by an exception handler of an enclosing `try` statement, after the `finally` block of the current `try` statement executes.

H.14 The reference goes out of scope. If the referenced object becomes unreachable, the object can be garbage collected.

Exercises

H.15 *(Exceptional Conditions)* List the various exceptional conditions that have occurred in programs throughout the appendices so far. List as many additional exceptional conditions as you can. For each of these, describe briefly how a program typically would handle the exception by using the exception-handling techniques discussed in this appendix. Typical exceptions include division by zero and array index out of bounds.

H.16 *(Exceptions and Constructor Failure)* Until this appendix, we've found dealing with errors detected by constructors to be a bit awkward. Explain why exception handling is an effective means for dealing with constructor failure.

H.17 *(Catching Exceptions with Superclasses)* Use inheritance to create an exception superclass (called `ExceptionA`) and exception subclasses `ExceptionB` and `ExceptionC`, where `ExceptionB` inherits from `ExceptionA` and `ExceptionC` inherits from `ExceptionB`. Write a program to demonstrate that the `catch` block for type `ExceptionA` catches exceptions of types `ExceptionB` and `ExceptionC`.

H.18 *(Catching Exceptions Using Class `Exception`)* Write a program that demonstrates how various exceptions are caught with

```
catch ( Exception exception )
```

This time, define classes `ExceptionA` (which inherits from class `Exception`) and `ExceptionB` (which inherits from class `ExceptionA`). In your program, create `try` blocks that throw exceptions of types `ExceptionA`, `ExceptionB`, `NullPointerException` and `IOException`. All exceptions should be caught with `catch` blocks specifying type `Exception`.

H.19 *(Usage of `finally` blocks)* Write a program that shows the usage of `finally` blocks. Your program should show the behavior of `finally` blocks in all the case scenarios mentioned in section H.6.

H.20 *(Constructor Failure)* Write a program that shows a constructor passing information about constructor failure to an exception handler. Define class `SomeClass`, which throws an `Exception` in the constructor. Your program should try to create an object of type `SomeClass` and catch the exception that's thrown from the constructor.

H.21 *(Rethrowing Exceptions)* Write a program that illustrates rethrowing an exception. Define methods `someMethod` and `someMethod2`. Method `someMethod2` should initially throw an exception. Method `someMethod` should call `someMethod2`, catch the exception and rethrow it. Call `someMethod` from method `main`, and catch the rethrown exception. Print the stack trace of this exception.

H.22 *(Catching Exceptions Using Outer Scopes)* Write a program showing that a method with its own `try` block does not have to catch every possible error generated within the `try`. Some exceptions can slip through to, and be handled in, other scopes.

I

GUI Components and Event Handling

Objectives

In this appendix you'll learn:

- How to use Java's cross-platform Nimbus look-and-feel.
- To build GUIs and handle events generated by user interactions with GUIs.
- To use nested classes and anonymous inner classes to implement event handlers.

Outline

I.1 Introduction

A **graphical user interface** (**GUI**) presents a user-friendly mechanism for interacting with an app. A GUI (pronounced "GOO-ee") gives an app a distinctive "look-and-feel." GUIs are built from **GUI components**, such as labels, buttons, textboxes, menus scrollbars and more. These are sometimes called controls or widgets—short for window gadgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. In this appendix, we introduce a few basic GUI components and how to respond to user interactions with them—a technique known as event handling. We also discuss *nested classes* and *anonymous inner classes*, which are commonly used for event handling in Java and Android apps.

I.2 Nimbus Look-and-Feel

In our screen captures, we use Java's elegant **Nimbus** cross-platform look-and-feel. There are three ways that you can use Nimbus:

1. Set it as the default for all Java apps that run on your computer.
2. Set it as the look-and-feel at the time that you launch an app by passing a command-line argument to the `java` command.
3. Set it as the look-and-feel programatically in your app.

We set Nimbus as the default for all Java apps. To do so, you must create a text file named `swing.properties` in the `lib` folder of both your JDK installation folder and your JRE installation folder. Place the following line of code in the file:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

For more information on locating these installation folders visit

```
bit.ly/JavaInstallationInstructions
```

In addition to the standalone JRE, there is a JRE nested in your JDK's installation folder. If you're using an IDE that depends on the JDK, you may also need to place the `swing.properties` file in the nested `jre` folder's `lib` folder.

If you prefer to select Nimbus on an app-by-app basis, place the following command-line argument after the `java` command and before the app's name when you run the app:

```
-Dswing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

I.3 Text Fields and an Introduction to Event Handling with Nested Classes

Normally, a user interacts with an app's GUI to indicate the tasks that the app should perform. For example, when you write an e-mail in an e-mail app, clicking the **Send** button tells the app to send the e-mail to the specified e-mail addresses. GUIs are **event driven**. When the user interacts with a GUI component, the interaction—known as an **event**—drives the program to perform a task. Some common user interactions that cause an app to perform a task include clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse. The code that performs a task in response to an event is called an **event handler**, and the overall process of responding to events is known as **event handling**.

Let's consider two GUI components that can generate events—**`JTextFields`** and **`JPasswordFields`** (package `javax.swing`). Class `JTextField` extends class **`JTextComponent`** (package `javax.swing.text`), which provides many features common to Swing's text-based components. Class `JPasswordField` extends `JTextField` and adds methods that are specific to processing passwords. Each of these components is a single-line area in which the user can enter text via the keyboard. Apps can also display text in a `JTextField` (see the output of Fig. I.2). A `JPasswordField` shows that characters are being typed as the user enters them, but hides the actual characters with an **echo character**, assuming that they represent a password that should remain known only to the user.

When the user types in a `JTextField` or a `JPasswordField`, then presses *Enter*, an event occurs. Our next example demonstrates how a program can perform a task in response to that event. The techniques shown here are applicable to all GUI components that generate events.

The app of Figs. I.1–I.2 uses classes `JTextField` and `JPasswordField` to create and manipulate four text fields. When the user types in one of the text fields, then presses *Enter*, the app displays a message dialog box containing the text the user typed. You can type only in the text field that's "in **focus**." When you click a component, it *receives the focus*. This is important, because the text field with the focus is the one that generates an event when you press *Enter*. In this example, you press *Enter* in the `JPasswordField`, the password is revealed. We begin by discussing the setup of the GUI, then discuss the event-handling code.

```java
// Fig. I.1: TextFieldFrame.java
// JTextFields and JPasswordFields.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

public class TextFieldFrame extends JFrame
{
```

Fig. I.1 | `JTextFields` and `JPasswordFields`. (Part 1 of 3.)

```
   private JTextField textField1; // text field with set size
   private JTextField textField2; // text field constructed with text
   private JTextField textField3; // text field with text and size
   private JPasswordField passwordField; // password field with text

   // TextFieldFrame constructor adds JTextFields to JFrame
   public TextFieldFrame()
   {
      super( "Testing JTextField and JPasswordField" );
      setLayout( new FlowLayout() ); // set frame layout

      // construct textfield with 10 columns
      textField1 = new JTextField( 10 );
      add( textField1 ); // add textField1 to JFrame

      // construct textfield with default text
      textField2 = new JTextField( "Enter text here" );
      add( textField2 ); // add textField2 to JFrame

      // construct textfield with default text and 21 columns
      textField3 = new JTextField( "Uneditable text field", 21 );
      textField3.setEditable( false ); // disable editing
      add( textField3 ); // add textField3 to JFrame

      // construct passwordfield with default text
      passwordField = new JPasswordField( "Hidden text" );
      add( passwordField ); // add passwordField to JFrame

      // register event handlers
      TextFieldHandler handler = new TextFieldHandler();
      textField1.addActionListener( handler );
      textField2.addActionListener( handler );
      textField3.addActionListener( handler );
      passwordField.addActionListener( handler );
   } // end TextFieldFrame constructor

   // private inner class for event handling
   private class TextFieldHandler implements ActionListener
   {
      // process text field events
      public void actionPerformed( ActionEvent event )
      {
         String string = ""; // declare string to display

         // user pressed Enter in JTextField textField1
         if ( event.getSource() == textField1 )
            string = String.format( "textField1: %s",
               event.getActionCommand() );

         // user pressed Enter in JTextField textField2
         else if ( event.getSource() == textField2 )
            string = String.format( "textField2: %s",
               event.getActionCommand() );
```

Fig. I.1 | JTextFields and JPasswordFields. (Part 2 of 3.)

```
66
67           // user pressed Enter in JTextField textField3
68           else if ( event.getSource() == textField3 )
69              string = String.format( "textField3: %s",
70                 event.getActionCommand() );
71
72           // user pressed Enter in JTextField passwordField
73           else if ( event.getSource() == passwordField )
74              string = String.format( "passwordField: %s",
75                 event.getActionCommand() );
76
77           // display JTextField content
78           JOptionPane.showMessageDialog( null, string );
79        } // end method actionPerformed
80     } // end private inner class TextFieldHandler
81  } // end class TextFieldFrame
```

Fig. I.1 | `JTextFields` and `JPasswordFields`. (Part 3 of 3.)

Lines 3–9 import the classes and interfaces we use in this example. Class `TextFieldFrame` extends `JFrame` and declares three `JTextField` variables and a `JPasswordField` vari-able (lines 13–16). Each of the corresponding text fields is instantiated and attached to the `TextFieldFrame` in the constructor (lines 19–47).

Specifying the Layout

When building a GUI, you must attach each GUI component to a container, such as a window created with a `JFrame`. Also, you typically must decide *where* to position each GUI component—known as specifying the layout. Java provides several **layout managers** that can help you position components.

Many IDEs provide GUI design tools in which you can specify components' exact sizes and locations in a visual manner by using the mouse; then the IDE will generate the GUI code for you. Such IDEs can greatly simplify GUI creation.

To ensure that our GUIs can be used with *any* IDE, we did *not* use an IDE to create the GUI code. We use Java's layout managers to size and position components. With the **`FlowLayout`** layout manager, components are placed on a container from left to right in the order in which they're added. When no more components can fit on the current line, they continue to display left to right on the next line. If the container is resized, a `FlowLayout` *reflows* the components, possibly with fewer or more rows based on the new container width. Every container has a default layout, which we're changing for `TextFieldFrame` to a `FlowLayout` (line 22). Method **`setLayout`** is inherited into class `TextFieldFrame` indirectly from class `Container`. The argument to the method must be an object of a class that implements the `LayoutManager` interface (e.g., `FlowLayout`). Line 22 creates a new `FlowLayout` object and passes its reference as the argument to `setLayout`.

Creating the GUI

Line 25 creates `textField1` with `10` columns of text. A text column's width in *pixels* is determined by the average width of a character in the text field's current font. When text is displayed in a text field and the text is wider than the field itself, a portion of the text at the right side is not visible. If you're typing in a text field and the cursor reaches the right edge, the text at the left edge is pushed off the left side of the field and is no longer visible. Users can use the left and right arrow keys to move through the complete text. Line 26 adds `textField1` to the `JFrame`.

Line 29 creates `textField2` with the initial text `"Enter text here"` to display in the text field. The width of the field is determined by the width of the default text specified in the constructor. Line 30 adds `textField2` to the `JFrame`.

Line 33 creates `textField3` and calls the `JTextField` constructor with two arguments—the default text `"Uneditable text field"` to display and the text field's width in columns (`21`). Line 34 uses method **`setEditable`** (inherited by `JTextField` from class `JTextComponent`) to make the text field *uneditable*—i.e., the user cannot modify the text in the field. Line 35 adds `textField3` to the `JFrame`.

Line 38 creates `passwordField` with the text `"Hidden text"` to display in the text field. The width of the field is determined by the width of the default text. When you execute the app, notice that the text is displayed as a string of asterisks. Line 39 adds `passwordField` to the `JFrame`.

Steps Required to Set Up Event Handling for a GUI Component
This example should display a message dialog containing the text from a text field when the user presses *Enter* in that text field. Before an app can respond to an event for a particular GUI component, you must:

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of the class from *Step 1* should be notified when the event occurs—known as **registering the event handler**.

Using a Nested Class to Implement an Event Handler
All the classes discussed so far were so-called **top-level classes**—that is, they were not declared inside another class. Java allows you to declare classes *inside* other classes—these are called **nested classes**. Nested classes can be `static` or non-`static`. Non-`static` nested classes are called **inner classes** and are frequently used to implement *event handlers*.

An inner-class object must be created by an object of the top-level class that contains the inner class. Each inner-class object *implicitly* has a reference to an object of its top-level class. The inner-class object is allowed to use this implicit reference to directly access all the variables and methods of the top-level class. A nested class that's `static` does not require an object of its top-level class and does not implicitly have a reference to an object of the top-level class.

Nested Class `TextFieldHandler`
The event handling in this example is performed by an object of the `private` inner class `TextFieldHandler` (lines 50–80). This class is `private` because it will be used only to create event handlers for the text fields in top-level class `TextFieldFrame`. As with other class members, *inner classes* can be declared `public`, `protected` or `private`. Since event handlers tend to be specific to the app in which they're defined, they're often implemented as `private` inner classes or as *anonymous inner classes* (Section I.7).

GUI components can generate many events in response to user interactions. Each event is represented by a class and can be processed only by the appropriate type of event handler. Normally, a component's supported events are described in the Java API documentation for that component's class and its superclasses. When the user presses *Enter* in a `JTextField` or `JPasswordField`, an **`ActionEvent`** (package `java.awt.event`) occurs. Such an event is processed by an object that implements the interface **`ActionListener`** (package `java.awt.event`). The information discussed here is available in the Java API documentation for classes `JTextField` and `ActionEvent`. Since `JPasswordField` is a subclass of `JTextField`, `JPasswordField` supports the same events.

To prepare to handle the events in this example, inner class `TextFieldHandler` implements interface `ActionListener` and declares the only method in that interface—`actionPerformed` (lines 53–79). This method specifies the tasks to perform when an `ActionEvent` occurs. So, inner class `TextFieldHandler` satisfies *Step 1* listed earlier in this section. We'll discuss the details of method `actionPerformed` shortly.

Registering the Event Handler for Each Text Field

In the `TextFieldFrame` constructor, line 42 creates a `TextFieldHandler` object and assigns it to variable `handler`. This object's `actionPerformed` method will be called automatically when the user presses *Enter* in any of the GUI's text fields. However, before this can occur, the program must register this object as the event handler for each text field. Lines 43–46 are the event-registration statements that specify `handler` as the event handler for the three `JTextField`s and the `JPasswordField`. The app calls `JTextField` method **`addActionListener`** to register the event handler for each component. This method receives as its argument an `ActionListener` object, which can be an object of any class that implements `ActionListener`. The object `handler` *is an* `ActionListener`, because class `TextFieldHandler` implements `ActionListener`. After lines 43–46 execute, the object `handler` **listens for events**. Now, when the user presses *Enter* in any of these four text fields, method `actionPerformed` (line 53–79) in class `TextFieldHandler` is called to handle the event. If an event handler is not registered for a particular text field, the event that occurs when the user presses *Enter* in that text field is **consumed**—i.e., it's simply ignored by the app.

Software Engineering Observation I.1

The event listener for an event must implement the appropriate event-listener interface.

Common Programming Error I.1

Forgetting to register an event-handler object for a particular GUI component's event type causes events of that type to be ignored.

Details of Class `TextFieldHandler`'s `actionPerformed` Method

In this example, we're using one event-handling object's `actionPerformed` method (lines 53–79) to handle the events generated by four text fields. Since we'd like to output the name of each text field's instance variable for demonstration purposes, we must determine which text field generated the event each time `actionPerformed` is called. The **event source** is the GUI component with which the user interacted. When the user presses *Enter* while one of the text fields or the password field *has the focus*, the system creates a unique `ActionEvent` object that contains information about the event that just occurred, such as the event source and the text in the text field. The system passes this `ActionEvent` object to the event listener's `actionPerformed` method. Line 55 declares the `String` that will be displayed. The variable is initialized with the **empty string**—a `String` containing no characters. The compiler requires the variable to be initialized in case none of the branches of the nested `if` in lines 58–75 executes.

`ActionEvent` method `getSource` (called in lines 58, 63, 68 and 73) returns a reference to the event source. The condition in line 58 asks, "Is the event source `textField1`?" This condition compares references with the == operator to determine if they refer to the same object. If they *both* refer to `textField1`, the user pressed *Enter* in `textField1`. Then, lines 59–60 create a `String` containing the message that line 78 displays in a message dialog. Line 60 uses `ActionEvent` method **`getActionCommand`** to obtain the text the user typed in the text field that generated the event.

In this example, we display the text of the password in the `JPasswordField` when the user presses *Enter* in that field. Sometimes it's necessary to programatically process the characters in a password. Class `JPasswordField` method **`getPassword`** returns the password's characters as an array of type `char`.

Class `TextFieldTest`

Class `TextFieldTest` (Fig. I.2) contains the `main` method that executes this app and displays an object of class `TextFieldFrame`. When you execute the app, even the uneditable `JTextField` (`textField3`) can generate an `ActionEvent`. To test this, click the text field to give it the focus, then press *Enter*. Also, the actual text of the password is displayed when you press *Enter* in the `JPasswordField`. Of course, you would normally not display the password!

This app used a single object of class `TextFieldHandler` as the event listener for four text fields. It's possible to declare several event-listener objects of the same type and register each object for a separate GUI component's event. This technique enables us to eliminate the `if...else` logic used in this example's event handler by providing separate event handlers for each component's events.

```
// Fig. I.2: TextFieldTest.java
// Testing TextFieldFrame.
import javax.swing.JFrame;

public class TextFieldTest
{
   public static void main( String[] args )
   {
      TextFieldFrame textFieldFrame = new TextFieldFrame();
      textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      textFieldFrame.setSize( 350, 100 ); // set frame size
      textFieldFrame.setVisible( true ); // display frame
   } // end main
} // end class TextFieldTest
```

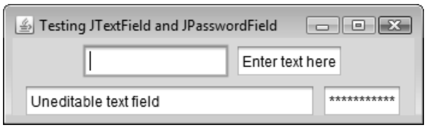
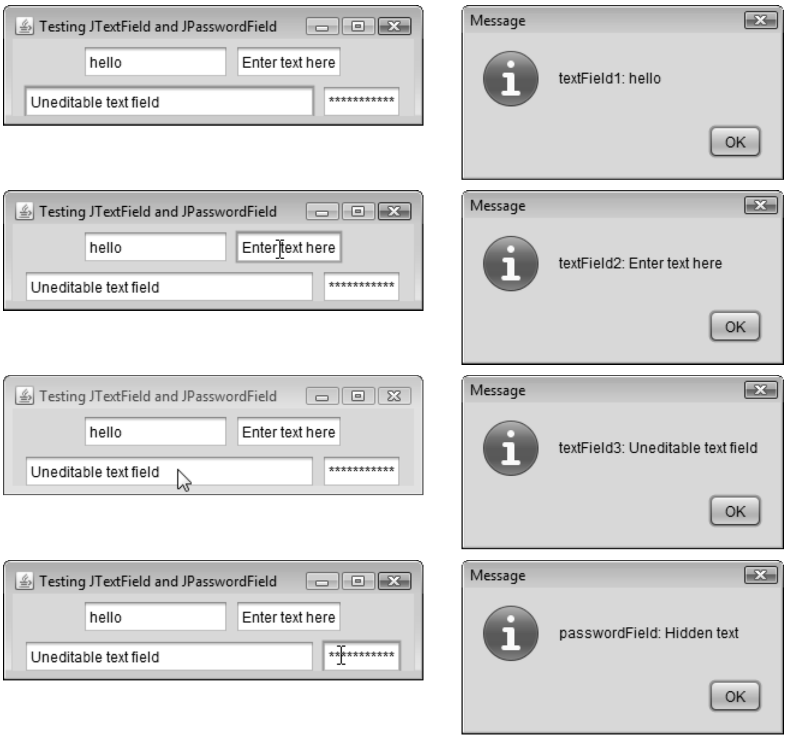


Fig. I.2 | Testing `TextFieldFrame`. (Part 1 of 2.)



textField1: hello

textField2: Enter text here

textField3: Uneditable text field

passwordField: Hidden text

Fig. I.2 | Testing `TextFieldFrame`. (Part 2 of 2.)

I.4 Common GUI Event Types and Listener Interfaces

In Section I.3, you learned that information about the event that occurs when the user presses *Enter* in a text field is stored in an `ActionEvent` object. Many different types of events can occur when the user interacts with a GUI. The event information is stored in an object of a class that extends `AWTEvent` (from package `java.awt`). Figure I.3 illustrates a hierarchy containing many event classes from the package **`java.awt.event`**. Additional event types are declared in package **`javax.swing.event`**.

Let's summarize the three parts to the event-handling mechanism that you saw in Section I.3—the *event source*, the *event object* and the *event listener*. The event source is the GUI component with which the user interacts. The event object encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The event listener is an object that's notified by the event source when an event occurs; in effect, it "listens" for an event, and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. This event-handling model is known as the **delegation event model**—an event's processing is delegated to an object (the event listener) in the app.

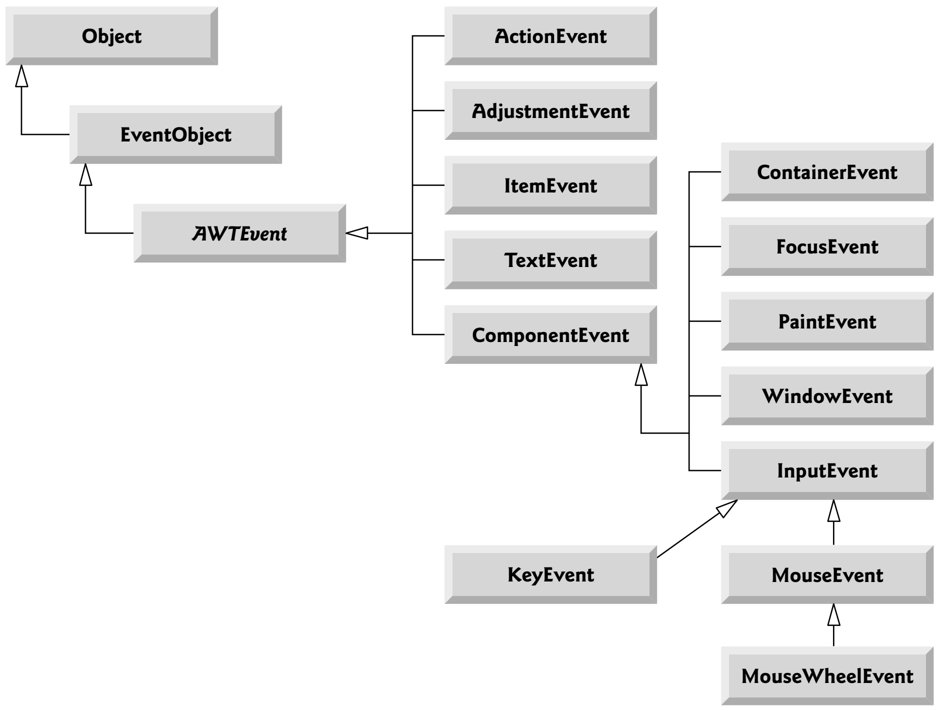


Fig. I.3 | Some event classes of package `java.awt.event`.

For each event-object type, there's typically a corresponding event-listener interface. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces.

Each event-listener interface specifies one or more event-handling methods that *must* be declared in the class that implements the interface. Recall from Section G.12 that any class which implements an interface must declare *all* the `abstract` methods of that interface; otherwise, the class is an `abstract` class and cannot be used to create objects.

When an event occurs, the GUI component with which the user interacted notifies its *registered listeners* by calling each listener's appropriate *event-handling method*. For example, when the user presses the *Enter* key in a `JTextField`, the registered listener's `actionPerformed` method is called. How did the event handler get registered? How does the GUI component know to call `actionPerformed` rather than another event-handling method? We answer these questions and diagram the interaction in the next section.

I.5 How Event Handling Works

Let's illustrate how the event-handling mechanism works, using `textField1` from the example of Fig. I.1. We have two remaining open questions from Section I.3:

1. How did the *event handler* get *registered*?
2. How does the GUI component know to call `actionPerformed` rather than some other event-handling method?

The first question is answered by the event registration performed in lines 43–46 of Fig. I.1. Figure I.4 diagrams `JTextField` variable `textField1`, `TextFieldHandler` variable `handler` and the objects to which they refer.

Registering Events

Every `JComponent` has an instance variable called `listenerList` that refers to an object of class **`EventListenerList`** (package `javax.swing.event`). Each object of a `JComponent` subclass maintains references to its registered listeners in the `listenerList`. For simplicity, we've diagramed `listenerList` as an array below the `JTextField` object in Fig. I.4.

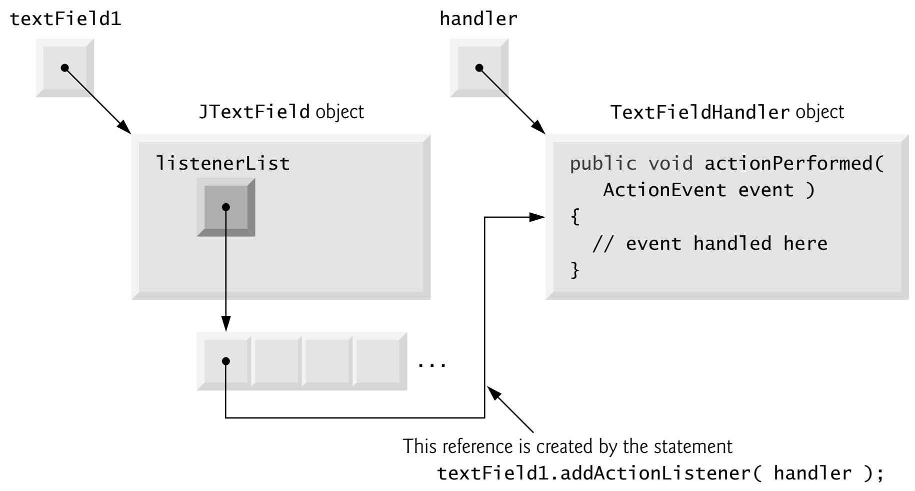

textField1
handler
JTextField object
TextFieldHandler object
listenerList
public void actionPerformed(ActionEvent event) { // event handled here }
...
This reference is created by the statement textField1.addActionListener(handler);

Fig. I.4 | Event registration for `JTextField` `textField1`.

When line 43 of Fig. I.1

```
    textField1.addActionListener( handler );
```

executes, a new entry containing a reference to the `TextFieldHandler` object is placed in `textField1`'s `listenerList`. Although not shown in the diagram, this new entry also includes the listener's type (in this case, `ActionListener`). Using this mechanism, each lightweight Swing GUI component maintains its own list of *listeners* that were *registered* to *handle* the component's *events*.

Event-Handler Invocation

The event-listener type is important in answering the second question: How does the GUI component know to call `actionPerformed` rather than another method? Every GUI component supports several *event types*, including **mouse events**, **key events** and others. When an event occurs, the event is **dispatched** only to the *event listeners* of the appropriate type. Dispatching is simply the process by which the GUI component calls an event-handling method on each of its listeners that are registered for the event type that occurred.

Each *event type* has one or more corresponding *event-listener interfaces*. For example, `ActionEvent`s are handled by `ActionListener`s, **`MouseEvent`s** by **`MouseListener`s** and

MouseMotionListeners, and **KeyEvents** by **KeyListeners**. When an event occurs, the GUI component receives (from the JVM) a unique ***event ID*** specifying the event type. The GUI component uses the event ID to decide the listener type to which the event should be dispatched and to decide which method to call on each listener object. For an `ActionEvent`, the event is dispatched to *every* registered `ActionListener`'s `actionPerformed` method (the only method in interface `ActionListener`). For a `MouseEvent`, the event is dispatched to *every* registered `MouseListener` or `MouseMotionListener`, depending on the mouse event that occurs. The `MouseEvent`'s event ID determines which of the several mouse event-handling methods are called. All these decisions are handled for you by the GUI components. All you need to do is register an event handler for the particular event type that your app requires, and the GUI component will ensure that the event handler's appropriate method gets called when the event occurs. We discuss other event types and event-listener interfaces as they're needed with each new component we introduce.

I.6 JButton

A **button** is a component the user clicks to trigger a specific action. A Java app can use several types of buttons, including **command buttons**, **checkboxes**, **toggle buttons** and **radio buttons**. Figure I.5 shows the inheritance hierarchy of the Swing buttons we cover in this appendix. As you can see, all the button types are subclasses of **`AbstractButton`** (package `javax.swing`), which declares the common features of Swing buttons. In this section, we concentrate on buttons that are typically used to initiate a command.

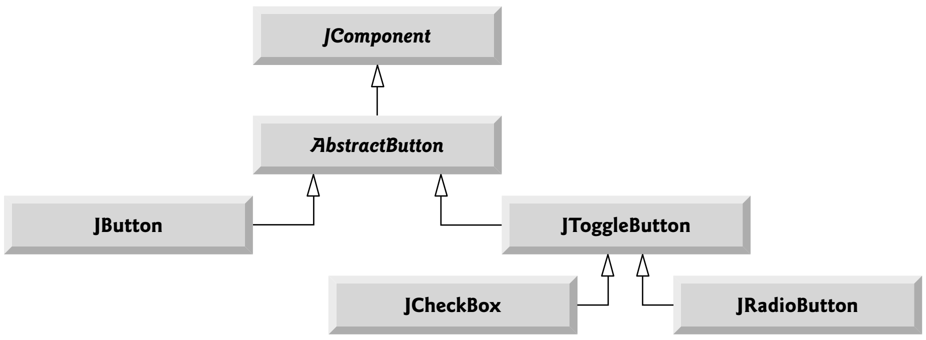


Fig. I.5 | Swing button hierarchy.

A command button (see Fig. I.7's output) generates an `ActionEvent` when the user clicks it. Command buttons are created with class **`JButton`**. The text on the face of a `JButton` is called a **button label**. A GUI can have many `JButtons`, but each button label should be unique in the portion of the GUI that's currently displayed.

Look-and-Feel Observation I.1

The text on buttons typically uses book-title capitalization.

Look-and-Feel Observation I.2

Having more than one `JButton` with the same label makes the `JButtons` ambiguous to the user. Provide a unique label for each button.

The app of Figs. I.6 and I.7 creates two `JButtons` and demonstrates that `JButtons` support the display of `Icons`. Event handling for the buttons is performed by a single instance of *inner class* `ButtonHandler` (lines 39–47).

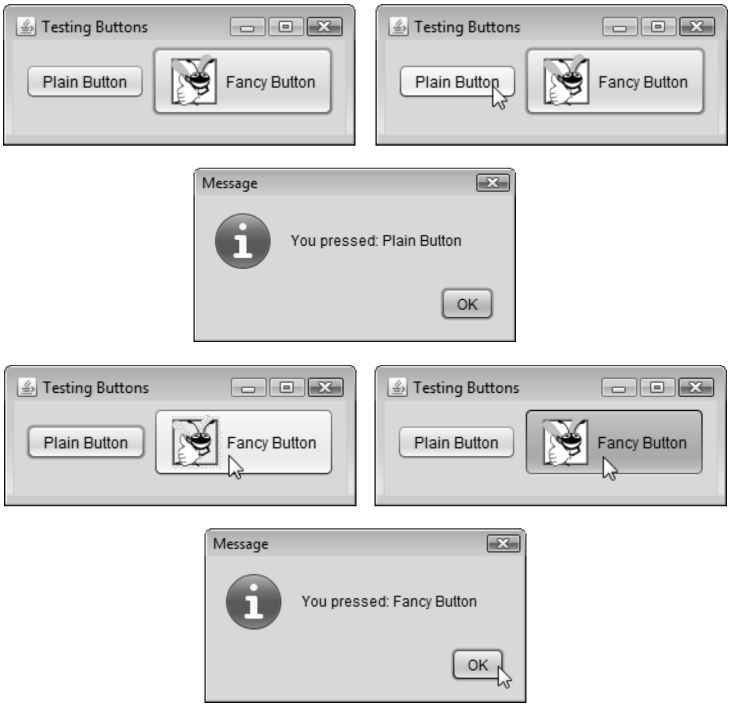
```
 1  // Fig. I.6: ButtonFrame.java
 2  // Command buttons and action events.
 3  import java.awt.FlowLayout;
 4  import java.awt.event.ActionListener;
 5  import java.awt.event.ActionEvent;
 6  import javax.swing.JFrame;
 7  import javax.swing.JButton;
 8  import javax.swing.Icon;
 9  import javax.swing.ImageIcon;
10  import javax.swing.JOptionPane;
11
12  public class ButtonFrame extends JFrame
13  {
14     private JButton plainJButton; // button with just text
15     private JButton fancyJButton; // button with icons
16
17     // ButtonFrame adds JButtons to JFrame
18     public ButtonFrame()
19     {
20        super( "Testing Buttons" );
21        setLayout( new FlowLayout() ); // set frame layout
22
23        plainJButton = new JButton( "Plain Button" ); // button with text
24        add( plainJButton ); // add plainJButton to JFrame
25
26        Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
27        Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
28        fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
29        fancyJButton.setRolloverIcon( bug2 ); // set rollover image
30        add( fancyJButton ); // add fancyJButton to JFrame
31
32        // create new ButtonHandler for button event handling
33        ButtonHandler handler = new ButtonHandler();
34        fancyJButton.addActionListener( handler );
35        plainJButton.addActionListener( handler );
36     } // end ButtonFrame constructor
37
38     // inner class for button event handling
39     private class ButtonHandler implements ActionListener
40     {
41        // handle button event
42        public void actionPerformed( ActionEvent event )
43        {
```

Fig. I.6 | Command buttons and action events. (Part 1 of 2.)

```
            JOptionPane.showMessageDialog( ButtonFrame.this, String.format(
               "You pressed: %s", event.getActionCommand() ) );
         } // end method actionPerformed
      } // end private inner class ButtonHandler
   } // end class ButtonFrame
```

Fig. I.6 | Command buttons and action events. (Part 2 of 2.)

```
// Fig. I.7: ButtonTest.java
// Testing ButtonFrame.
import javax.swing.JFrame;

public class ButtonTest
{
   public static void main( String[] args )
   {
      ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
      buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      buttonFrame.setSize( 275, 110 ); // set frame size
      buttonFrame.setVisible( true ); // display frame
   } // end main
} // end class ButtonTest
```



You pressed: Plain Button

You pressed: Fancy Button

Fig. I.7 | Testing ButtonFrame.

Lines 14–15 of Fig. I.6 declare JButton variables plainJButton and fancyJButton. The corresponding objects are instantiated in the constructor. Line 23 creates plainJButton with the button label "Plain Button". Line 24 adds the JButton to the JFrame.

A JButton can display an Icon. To provide the user with an extra level of visual interaction with the GUI, a JButton can also have a **rollover Icon**—an Icon that's displayed when the user positions the mouse over the JButton. The icon on the JButton changes as the mouse moves in and out of the JButton's area on the screen. Lines 26–27 (Fig. I.6) create two ImageIcon objects that represent the default Icon and rollover Icon for the JButton created at line 28. Both statements assume that the image files are stored in the same directory as the app. Images are commonly placed in the same directory as the app or a subdirectory like images). These image files have been provided for you with the example.

Line 28 creates fancyButton with the text "Fancy Button" and the icon bug1. By default, the text is displayed to the right of the icon. Line 29 uses **setRolloverIcon** (inherited from class AbstractButton) to specify the image displayed on the JButton when the user positions the mouse over it. Line 30 adds the JButton to the JFrame.

Look-and-Feel Observation I.3

Because class AbstractButton supports displaying text and images on a button, all subclasses of AbstractButton also support displaying text and images.

Look-and-Feel Observation I.4

Using rollover icons for JButtons provides users with visual feedback indicating that when they click the mouse while the cursor is positioned over the JButton, an action will occur.

JButtons, like JTextFields, generate ActionEvents that can be processed by any ActionListener object. Lines 33–35 create an object of private *inner class* ButtonHandler and use addActionListener to *register* it as the *event handler* for each JButton. Class ButtonHandler (lines 39–47) declares actionPerformed to display a message dialog box containing the label for the button the user pressed. For a JButton event, ActionEvent method getActionCommand returns the label on the JButton.

Accessing the this Reference in an Object of a Top-Level Class From a Nested Class

When you execute this app and click one of its buttons, notice that the message dialog that appears is centered over the app's window. This occurs because the call to JOptionPane method showMessageDialog (lines 44–45 of Fig. I.6) uses ButtonFrame.this rather than null as the first argument. When this argument is not null, it represents the so-called *parent GUI component* of the message dialog (in this case the app window is the parent component) and enables the dialog to be centered over that component when the dialog is displayed. ButtonFrame.this represents the this reference of the object of top-level class ButtonFrame.

Software Engineering Observation I.2

When used in an inner class, keyword this refers to the current inner-class object being manipulated. An inner-class method can use its outer-class object's this by preceding this with the outer-class name and a dot, as in ButtonFrame.this.

I.7 JComboBox; Using an Anonymous Inner Class for Event Handling

A combo box (sometimes called a **drop-down list**) enables the user to select one item from a list (Fig. I.9). Combo boxes are implemented with class **JComboBox**, which extends class JComponent. JComboBoxes generate ItemEvents just as JCheckBoxes and JRadioButtons do. This example also demonstrates a special form of inner class that's used frequently in event handling. The app (Figs. I.8–I.9) uses a JComboBox to provide a list of four image-file names from which the user can select one image to display. When the user selects a name, the app displays the corresponding image as an Icon on a JLabel. Class ComboBoxTest (Fig. I.9) contains the main method that executes this app. The screen captures for this app show the JComboBox list after the selection was made to illustrate which image-file name was selected.

Lines 19–23 (Fig. I.8) declare and initialize array icons with four new ImageIcon objects. String array names (lines 17–18) contains the names of the four image files that are stored in the same directory as the app.

```
 1  // Fig. I.8: ComboBoxFrame.java
 2  // JComboBox that displays a list of image names.
 3  import java.awt.FlowLayout;
 4  import java.awt.event.ItemListener;
 5  import java.awt.event.ItemEvent;
 6  import javax.swing.JFrame;
 7  import javax.swing.JLabel;
 8  import javax.swing.JComboBox;
 9  import javax.swing.Icon;
10  import javax.swing.ImageIcon;
11
12  public class ComboBoxFrame extends JFrame
13  {
14     private JComboBox imagesJComboBox; // combobox to hold names of icons
15     private JLabel label; // label to display selected icon
16
17     private static final String[] names =
18        { "bug1.gif", "bug2.gif",  "travelbug.gif", "buganim.gif" };
19     private Icon[] icons = {
20        new ImageIcon( getClass().getResource( names[ 0 ] ) ),
21        new ImageIcon( getClass().getResource( names[ 1 ] ) ),
22        new ImageIcon( getClass().getResource( names[ 2 ] ) ),
23        new ImageIcon( getClass().getResource( names[ 3 ] ) ) };
24
25     // ComboBoxFrame constructor adds JComboBox to JFrame
26     public ComboBoxFrame()
27     {
28        super( "Testing JComboBox" );
29        setLayout( new FlowLayout() ); // set frame layout
30
31        imagesJComboBox = new JComboBox( names ); // set up JComboBox
32        imagesJComboBox.setMaximumRowCount( 3 ); // display three rows
```

Fig. I.8 | JComboBox that displays a list of image names. (Part 1 of 2.)

```

      imagesJComboBox.addItemListener(
         new ItemListener() // anonymous inner class
         {
            // handle JComboBox event
            public void itemStateChanged( ItemEvent event )
            {
               // determine whether item selected
               if ( event.getStateChange() == ItemEvent.SELECTED )
                  label.setIcon( icons[
                     imagesJComboBox.getSelectedIndex() ] );
            } // end method itemStateChanged
         } // end anonymous inner class
      ); // end call to addItemListener

      add( imagesJComboBox ); // add combobox to JFrame
      label = new JLabel( icons[ 0 ] ); // display first icon
      add( label ); // add label to JFrame
   } // end ComboBoxFrame constructor
} // end class ComboBoxFrame
```

Fig. I.8 | JComboBox that displays a list of image names. (Part 2 of 2.)

```
// Fig. I.9: ComboBoxTest.java
// Testing ComboBoxFrame.
import javax.swing.JFrame;

public class ComboBoxTest
{
   public static void main( String[] args )
   {
      ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
      comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      comboBoxFrame.setSize( 350, 150 ); // set frame size
      comboBoxFrame.setVisible( true ); // display frame
   } // end main
} // end class ComboBoxTest
```

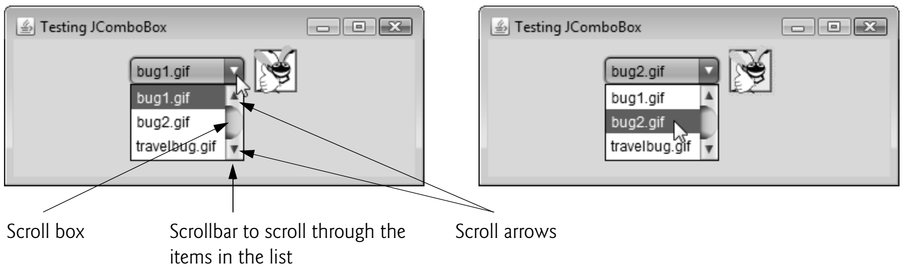
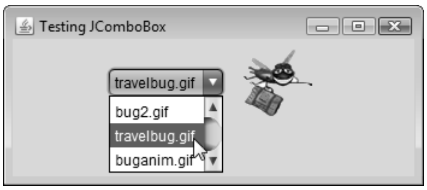


Fig. I.9 | Testing ComboBoxFrame. (Part 1 of 2.)



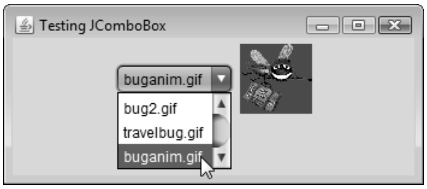


Fig. I.9 | Testing `ComboBoxFrame`. (Part 2 of 2.)

At line 31, the constructor initializes a `JComboBox` object with the `Strings` in array `names` as the elements in the list. Each item in the list has an **index**. The first item is added at index 0, the next at index 1 and so forth. The first item added to a `JComboBox` appears as the currently selected item when the `JComboBox` is displayed. Other items are selected by clicking the `JComboBox`, then selecting an item from the list that appears.

Line 32 uses `JComboBox` method **`setMaximumRowCount`** to set the maximum number of elements that are displayed when the user clicks the `JComboBox`. If there are additional items, the `JComboBox` provides a **scrollbar** (see the first screen) that allows the user to scroll through all the elements in the list. The user can click the **scroll arrows** at the top and bottom of the scrollbar to move up and down through the list one element at a time, or else drag the **scroll box** in the middle of the scrollbar up and down. To drag the scroll box, position the mouse cursor on it, hold the mouse button down and move the mouse. In this example, the drop-down list is too short to drag the scroll box, so you can click the up and down arrows or use your mouse's wheel to scroll through the four items in the list.

Look-and-Feel Observation I.5

Set the maximum row count for a `JComboBox` to a number of rows that prevents the list from expanding outside the bounds of the window in which it's used.

Line 48 attaches the `JComboBox` to the `ComboBoxFrame`'s `FlowLayout` (set in line 29). Line 49 creates the `JLabel` that displays `ImageIcons` and initializes it with the first `Image-Icon` in array `icons`. Line 50 attaches the `JLabel` to the `ComboBoxFrame`'s `FlowLayout`.

Using an Anonymous Inner Class for Event Handling

Lines 34–46 are one statement that declares the event listener's class, creates an object of that class and registers it as the listener for `imagesJComboBox`'s `ItemEvents`. This event-listener object is an instance of an **anonymous inner class**—an inner class that's declared without a name and typically appears inside a method declaration. *As with other inner classes, an anonymous inner class can access its top-level class's members.* However, an anonymous inner class has limited access to the local variables of the method in which it's declared. Since an anonymous inner class has no name, one object of the class must be created at the point where the class is declared (starting at line 35).

Software Engineering Observation I.3

An anonymous inner class declared in a method can access the instance variables and methods of the top-level class object that declared it, as well as the method's `final` local variables, but cannot access the method's non-`final` local variables.

Lines 34–46 are a call to `imagesJComboBox`'s `addItemListener` method. The argument to this method must be an object that *is an* `ItemListener` (i.e., any object of a class that implements `ItemListener`). Lines 35–45 are a class-instance creation expression that declares an anonymous inner class and creates one object of that class. A reference to that object is then passed as the argument to `addItemListener`. The syntax `ItemListener()` after `new` begins the declaration of an anonymous inner class that implements interface `ItemListener`. This is similar to beginning a class declaration with

```
    public class MyHandler implements ItemListener
```

The opening left brace at 36 and the closing right brace at line 45 delimit the body of the anonymous inner class. Lines 38–44 declare the `ItemListener`'s `itemStateChanged` method. When the user makes a selection from `imagesJComboBox`, this method sets `label`'s `Icon`. The `Icon` is selected from array `icons` by determining the index of the selected item in the `JComboBox` with method **`getSelectedIndex`** in line 43. For each item selected from a `JComboBox`, another item is first deselected—so two `ItemEvents` occur when an item is selected. We wish to display only the icon for the item the user just selected. For this reason, line 41 determines whether `ItemEvent` method **`getStateChange`** returns `ItemEvent.SELECTED`. If so, lines 42–43 set `label`'s icon.

Software Engineering Observation I.4

Like any other class, when an anonymous inner class implements an interface, the class must implement every method in the interface.

The syntax shown in lines 35–45 for creating an event handler with an anonymous inner class is similar to the code that would be generated by a Java IDE. Typically, an IDE enables you to design a GUI visually, then it generates code that implements the GUI. You simply insert statements in the event-handling methods that declare how to handle each event.

I.8 Adapter Classes

Many event-listener interfaces, such as `MouseListener` and `MouseMotionListener`, contain multiple methods. It's not always desirable to declare every method in an event-listener interface. For instance, an app may need only the `mouseClicked` handler from `MouseListener` or the `mouseDragged` handler from `MouseMotionListener`. Interface `WindowListener` specifies seven window event-handling methods. For many of the listener interfaces that have multiple methods, packages `java.awt.event` and `javax.swing.event` provide event-listener adapter classes. An **adapter class** implements an interface and provides a default implementation (with an empty method body) of each method in the interface. You can extend an adapter class to inherit the default implementation of every method and subsequently override only the method(s) you need for event handling.

Software Engineering Observation I.5

When a class implements an interface, the class has an is-a *relationship with that interface. All direct and indirect subclasses of that class inherit this interface. Thus, an object of a class that extends an event-adapter class* is an *object of the corresponding event-listener type (e.g., an object of a subclass of* `MouseAdapter` *is a* `MouseListener`*).*

I.9 Wrap-Up

In this appendix, you learned about a few Java GUI components and how to implement event handlers using nested classes and anonymous inner classes. You saw the special relationship between an inner-class object and an object of its top-level class. You also learned how to create apps that execute in their own windows. We discussed class `JFrame` and components that enable a user to interact with an app.

Self-Review Exercises

I.1 Fill in the blanks in each of the following statements:

a) A(n) ________ arranges GUI components in a `Container`.
b) The `add` method for attaching GUI components is a method of class ________.
c) GUI is an acronym for ________.
d) Method ________ is used to specify the layout manager for a container.

I.2 Specify whether the folowing statement is *true* or *false* and if *false*, explain why: Inner classes are not allowed to access the members of the enclosing class.

Answers to Self-Review Exercises

I.1 a) layout manager. b) `Container`. c) graphical user interface. d) `setLayout`.

I.2 False. Inner classes have access to all members of the enclosing class declaration.

Exercises

I.3 *(Temperature Conversion)* Write a temperature-conversion app that converts from Fahrenheit to Celsius. The Fahrenheit temperature should be entered from the keyboard (via a `JTextField`). A `JLabel` should be used to display the converted temperature. Use the following formula for the conversion:

$$Celsius = \frac{5}{9} \times (Fahrenheit - 32)$$

I.4 *(Temperature-Conversion Modification)* Enhance the temperature-conversion app of Exercise I.3 by adding the Kelvin temperature scale. The app should also allow the user to make conversions between any two scales. Use the following formula for the conversion between Kelvin and Celsius (in addition to the formula in Exercise I.3):

$$Kelvin = Celsius + 273.15$$

I.5 *(Guess-the-Number Game)* Write an app that plays "guess the number" as follows: Your app chooses the number to be guessed by selecting an integer at random in the range 1–1000. The app then displays the following in a label:

```
I have a number between 1 and 1000. Can you guess my number?
Please enter your first guess.
```

A `JTextField` should be used to input the guess. As each guess is input, the background color should change to either red or blue. Red indicates that the user is getting "warmer," and blue, "colder." A `JLabel` should display either `"Too High"` or `"Too Low"` to help the user zero in. When the user gets the correct answer, `"Correct!"` should be displayed, and the `JTextField` used for input should be changed to be uneditable. A `JButton` should be provided to allow the user to play the game again. When the `JButton` is clicked, a new random number should be generated and the input `JTextField` changed to be editable.

J Other Java Topics

Objectives

In this appendix you'll:

- Learn what collections are.
- Use class `Arrays` for array manipulations.
- Understand how type-wrapper classes enable programs to process primitive data values as objects.
- Use prebuilt generic data structures from the collections framework.
- Use iterators to "walk through" a collection.
- Learn fundamental file- and stream-processing concepts.
- What threads are and why they're useful.
- How threads enable you to manage concurrent activities.
- To create and execute `Runnable`s.
- Fundamentals of thread synchronization.
- How multiple threads can update Swing GUI components in a thread-safe manner.

Outline

J.1 Introduction

This appendix presents several additional topics to support the Android portion of the book. Sections J.2–J.9 present an overview of the Java collections framework and several examples of working with various collections that we use in our Android apps. Sections J.10–J.12 introduce file and stream concepts, overview method of class File and discuss object-serialization for writing entire objects to streams and reading entire objects from streams. Finally, Sections J.13–J.17 present the fundamentals of multithreading.

J.2 Collections Overview

Section E.12 introducted the generic `ArrayList` collection—a resizable array-like data structure that stores references to objects of a type that you specify when you create the `ArrayList`. We now continue our discussion of the Java **collections framework**, which contains many other prebuilt generic data structures and various methods for manipulating them. We focus on those that are used in the Android chapters of this book and those that have close parallels in the Android APIs. For complete details of the collections framework, visit

```
docs.oracle.com/javase/6/docs/technotes/guides/collections/
```

A **collection** is a data structure—actually, an object—that can hold references to other objects. Usually, collections contain references to objects that are all of the same type. The collections-framework interfaces declare the operations to be performed generically on various types of collections. Figure J.1 lists some of the interfaces of the collections framework. Several implementations of these interfaces are provided within the framework. You may also provide implementations specific to your own requirements.

Because you specify the type to store in a collection at compile time, generic collections provide compile-time type safety that allows the compiler to catch attempts to use invalid types. For example, you cannot store `Employees` in a collection of `Strings`. Some

Interface	Description
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does not contain duplicates.
`List`	An ordered collection that can contain duplicate elements.
`Map`	A collection that associates keys to values and cannot contain duplicate keys.
`Queue`	Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

Fig. J.1 | Some collections-framework interfaces.

examples of collections are the cards you hold in a card game, your favorite songs stored in your computer, the members of a sports team and the real-estate records in your local registry of deeds (which map book numbers and page numbers to property owners).

J.3 Type-Wrapper Classes for Primitive Types

Each primitive type (listed in Appendix L) has a corresponding **type-wrapper class** in package `java.lang`. These classes are called **`Boolean`**, **`Byte`**, **`Character`**, **`Double`**, **`Float`**, **`Integer`**, **`Long`** and **`Short`**. These enable you to manipulate primitive-type values as objects. Java's reusable data structures manipulate and share *objects*—they cannot manipulate variables of primitive types. However, they can manipulate objects of the type-wrapper classes, because every class ultimately derives from `Object`.

Each of the numeric type-wrapper classes—`Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`—extends class `Number`. Also, the type-wrapper classes are `final` classes, so you cannot extend them.

Primitive types do not have methods, so the methods related to a primitive type are located in the corresponding type-wrapper class (e.g., method `parseInt`, which converts a `String` to an `int` value, is located in class `Integer`). If you need to manipulate a primitive value in your program, first refer to the documentation for the type-wrapper classes—the method you need might already be declared.

Autoboxing and Auto-Unboxing

Java provides *boxing* and *unboxing conversions* to automatically convert between primitive-type values and type-wrapper objects. A **boxing conversion** converts a value of a primitive type to an object of the corresponding type-wrapper class. An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding primitive type. These conversions are performed automatically (called **autoboxing** and **auto-unboxing**), allowing primitive-type values to be used where type-wrapper objects are expected and vice versa.

J.4 Interface `Collection` and Class `Collections`

Interface **`Collection`** is the root interface in the collection hierarchy from which interfaces `Set`, `Queue` and `List` are derived. Interface **`Set`** defines a collection that does not contain duplicates. Interface **`Queue`** defines a collection that represents a waiting line—typically, insertions are made at the back of a queue and deletions from the front, though other orders can be specified. We discuss `Queue` and `Set` in Sections J.7—J.8. Interface `Collection` contains **bulk operations** (i.e., operations performed on an entire collection) for operations such as adding, clearing and comparing objects (or elements) in a collection. A `Collection` can also be converted to an array. In addition, interface `Collection` provides a method that returns an **`Iterator`** object, which allows a program to walk through the collection and remove elements from it during the iteration. We discuss class `Iterator` in Section J.5.1. Other methods of interface `Collection` enable a program to determine a collection's size and whether a collection is empty. Class **`Collections`** provides `static` methods that search, sort and perform other operations on collections. Section J.6 discusses the methods that are available in class `Collections`.

Software Engineering Observation J.1

Most collection implementations provide a constructor that takes a `Collection` argument, thereby allowing a new collection to be constructed containing the elements of the specified collection.

J.5 Lists

A `List` is an ordered `Collection` that can contain duplicate elements. Like array indices, `List` indices are zero based (i.e., the first element's index is zero). In addition to the methods inherited from `Collection`, interface `List` provides methods for manipulating elements via their indices, manipulating a specified range of elements, searching for elements and obtaining a **`ListIterator`** to access the elements.

Interface `List` is implemented by several classes, including **`ArrayList`** (introduced in Appendix E) and **`LinkedList`**. Class `ArrayList` is a resizable-array implementation of `List`. Inserting an element between existing elements of an `ArrayList` is an *inefficient* operation—all elements after the new one must be moved out of the way, which could be an expensive operation in a collection with a large number of elements. A `LinkedList` enables efficient insertion (or removal) of elements in the middle of a collection. The following two subsections demonstrate various `List` and `Collection` capabilities.

J.5.1 `ArrayList` and `Iterator`

Figure J.2 uses an `ArrayList` (introduced in Section E.12) to demonstrate several capabilities of interface `Collection`. The program places two `Color` arrays in `ArrayList`s and uses an `Iterator` to remove elements in the second `ArrayList` collection from the first.

```java
// Fig. J.2: CollectionTest.java
// Collection interface demonstrated via an ArrayList object.
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionTest
{
```

Fig. J.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 1 of 2.)

```
   public static void main( String[] args )
   {
      // add elements in colors array to list
      String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
      List< String > list = new ArrayList< String >();

      for ( String color : colors )
         list.add( color ); // adds color to end of list

      // add elements in removeColors array to removeList
      String[] removeColors = { "RED", "WHITE", "BLUE" };
      List< String > removeList = new ArrayList< String >();

      for ( String color : removeColors )
         removeList.add( color );

      // output list contents
      System.out.println( "ArrayList: " );

      for ( int count = 0; count < list.size(); count++ )
         System.out.printf( "%s ", list.get( count ) );

      // remove from list the colors contained in removeList
      removeColors( list, removeList );

      // output list contents
      System.out.println( "\n\nArrayList after calling removeColors: " );

      for ( String color : list )
         System.out.printf( "%s ", color );
   } // end main

   // remove colors specified in collection2 from collection1
   private static void removeColors( Collection< String > collection1,
      Collection< String > collection2 )
   {
      // get iterator
      Iterator< String > iterator = collection1.iterator();

      // loop while collection has items
      while ( iterator.hasNext() )
      {
         if ( collection2.contains( iterator.next() ) )
            iterator.remove(); // remove current Color
      } // end while
   } // end method removeColors
} // end class CollectionTest
```

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

Fig. J.2 | `Collection` interface demonstrated via an `ArrayList` object. (Part 2 of 2.)

Lines 13 and 20 declare and initialize `String` arrays `colors` and `removeColors`. Lines 14 and 21 create `ArrayList<String>` objects and assign their references to `List<String>` variables `list` and `removeList`, respectively. We refer to the `ArrayList`s in this example via `List` variables. This makes our code more flexible and easier to modify. If we later decide that `LinkedList`s would be more appropriate, we'll need to modify only lines 14 and 21 where we created the `ArrayList` objects.

Lines 16–17 populate `list` with `String`s stored in array `colors`, and lines 23–24 populate `removeList` with `String`s stored in array `removeColors` using **`List` method `add`**. Lines 29–30 output each element of `list`. Line 29 calls **`List` method `size`** to get the number of elements in the `ArrayList`. Line 30 uses **`List` method `get`** to retrieve individual element values. Lines 29–30 also could have used the enhanced `for` statement (which we'll demonstrate with collections in other examples).

Line 33 calls method `removeColors` (lines 43–55), passing `list` and `removeList` as arguments. Method `removeColors` deletes the `String`s in `removeList` from the `String`s in `list`. Lines 38–39 print `list`'s elements after `removeColors` completes its task.

Method `removeColors` declares two `Collection<String>` parameters (lines 43–44) that allow any two `Collection`s containing strings to be passed as arguments to this method. The method accesses the elements of the first `Collection` (`collection1`) via an `Iterator`. Line 47 calls `Collection` method **`iterator`** to get an `Iterator` for the `Collection`. Interfaces `Collection` and `Iterator` are generic types. The loop-continuation condition (line 50) calls `Iterator` method **`hasNext`** to determine whether the `Collection` contains more elements. Method `hasNext` returns `true` if another element exists and `false` otherwise.

The `if` condition in line 52 calls **`Iterator` method `next`** to obtain a reference to the next element, then uses method **`contains`** of the second `Collection` (`collection2`) to determine whether `collection2` contains the element returned by `next`. If so, line 53 calls **`Iterator` method `remove`** to remove the element from the `Collection` `collection1`.

Common Programming Error J.1

If a collection is modified after an iterator is created for that collection, the iterator immediately becomes invalid—operations performed with the iterator after this point throw `ConcurrentModificationException`s. For this reason, iterators are said to be "fail fast."

J.5.2 `LinkedList`

Figure J.3 demonstrates various operations on `LinkedList`s. The program creates two `LinkedList`s of `String`s. The elements of one `List` are added to the other. Then all the `String`s are converted to uppercase, and a range of elements is deleted.

```java
// Fig. J.3: ListTest.java
// Lists, LinkedLists and ListIterators.
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;

public class ListTest
{
```

Fig. J.3 | `Lists`, `LinkedLists` and `ListIterators`. (Part 1 of 3.)

```java
   public static void main( String[] args )
   {
      // add colors elements to list1
      String[] colors =
         { "black", "yellow", "green", "blue", "violet", "silver" };
      List< String > list1 = new LinkedList< String >();

      for ( String color : colors )
         list1.add( color );

      // add colors2 elements to list2
      String[] colors2 =
         { "gold", "white", "brown", "blue", "gray", "silver" };
      List< String > list2 = new LinkedList< String >();

      for ( String color : colors2 )
         list2.add( color );

      list1.addAll( list2 ); // concatenate lists
      list2 = null; // release resources
      printList( list1 ); // print list1 elements

      convertToUppercaseStrings( list1 ); // convert to uppercase string
      printList( list1 ); // print list1 elements

      System.out.print( "\nDeleting elements 4 to 6..." );
      removeItems( list1, 4, 7 ); // remove items 4-6 from list
      printList( list1 ); // print list1 elements
      printReversedList( list1 ); // print list in reverse order
   } // end main

   // output List contents
   private static void printList( List< String > list )
   {
      System.out.println( "\nlist: " );

      for ( String color : list )
         System.out.printf( "%s ", color );

      System.out.println();
   } // end method printList

   // locate String objects and convert to uppercase
   private static void convertToUppercaseStrings( List< String > list )
   {
      ListIterator< String > iterator = list.listIterator();

      while ( iterator.hasNext() )
      {
         String color = iterator.next(); // get item
         iterator.set( color.toUpperCase() ); // convert to upper case
      } // end while
   } // end method convertToUppercaseStrings
```

Fig. J.3 | Lists, LinkedLists and ListIterators. (Part 2 of 3.)

```
62
63       // obtain sublist and use clear method to delete sublist items
64       private static void removeItems( List< String > list,
65          int start, int end )
66       {
67          list.subList( start, end ).clear(); // remove items
68       } // end method removeItems
69
70       // print reversed list
71       private static void printReversedList( List< String > list )
72       {
73          ListIterator< String > iterator = list.listIterator( list.size() );
74
75          System.out.println( "\nReversed List:" );
76
77          // print list in reverse order
78          while ( iterator.hasPrevious() )
79             System.out.printf( "%s ", iterator.previous() );
80       } // end method printReversedList
81    } // end class ListTest
```

```
list:
black yellow green blue violet silver gold white brown blue gray silver

list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER

Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

Fig. J.3 | `Lists`, `LinkedLists` and `ListIterators`. (Part 3 of 3.)

Lines 14 and 22 create `LinkedLists` `list1` and `list2` of type `String`. `LinkedList` is a generic class that has one type parameter for which we specify the type argument `String` in this example. Lines 16–17 and 24–25 call `List` method `add` to append elements from arrays `colors` and `colors2` to the end of `list1` and `list2`, respectively.

Line 27 calls **`List` method `addAll`** to append all elements of `list2` to the end of `list1`. Line 28 sets `list2` to `null`, so the `LinkedList` to which `list2` referred can be garbage collected. Line 29 calls method `printList` (lines 41–49) to output `list1`'s contents. Line 31 calls method `convertToUppercaseStrings` (lines 52–61) to convert each `String` element to uppercase, then line 32 calls `printList` again to display the modified `Strings`. Line 35 calls method `removeItems` (lines 64–68) to remove the elements starting at index `4` up to, but not including, index `7` of the list. Line 37 calls method `printReversedList` (lines 71–80) to print the list in reverse order.

Method `convertToUppercaseStrings`

Method `convertToUppercaseStrings` (lines 52–61) changes lowercase `String` elements in its `List` argument to uppercase `Strings`. Line 54 calls **`List` method `listIterator`** to get the `List`'s **bidirectional iterator** (i.e., one that can traverse a `List` backward or forward).

ListIterator is also a generic class. In this example, the ListIterator references String objects, because method listIterator is called on a List of Strings. Line 56 calls method hasNext to determine whether the List contains another element. Line 58 gets the next String in the List. Line 59 calls **String method toUpperCase** to get an uppercase version of the String and calls **ListIterator method set** to replace the current String to which iterator refers with the String returned by method toUpperCase. Like method toUpperCase, **String method toLowerCase** returns a lowercase version of the String.

Method removeItems

Method removeItems (lines 64–68) removes a range of items from the list. Line 67 calls **List method subList** to obtain a portion of the List (called a **sublist**). This is a so-called **range-view method**, which enables the program to view a portion of the list. The sublist is simply a view into the List on which subList is called. Method subList takes as arguments the beginning and ending index for the sublist. The ending index is not part of the range of the sublist. In this example, line 35 passes 4 for the beginning index and 7 for the ending index to subList. The sublist returned is the set of elements with indices 4 through 6. Next, the program calls **List method clear** on the sublist to remove the elements of the sublist from the List. Any changes made to a sublist are also made to the original List.

Method printReversedList

Method printReversedList (lines 71–80) prints the list backward. Line 73 calls List method listIterator with the starting position as an argument (in our case, the last element in the list) to get a bidirectional iterator for the list. **List method size** returns the number of items in the List. The while condition (line 78) calls **ListIterator's hasPrevious method** to determine whether there are more elements while traversing the list backward. Line 79 calls **ListIterator's previous method** to get the previous element from the list and outputs it to the standard output stream.

J.5.3 Views into Collections and Arrays Method asList

An important feature of the collections framework is the ability to manipulate the elements of one collection type (such as a set) through a different collection type (such as a list), regardless of the collection's internal implementation. The set of public methods through which collections are manipulated is called a **view**.

Class Arrays provides static method **asList** to view an array (sometimes called the **backing array**) as a **List** collection. A List view allows you to manipulate the array as if it were a list. This is useful for adding the elements in an array to a collection and for sorting array elements. The next example demonstrates how to create a LinkedList with a List view of an array, because we cannot pass the array to a LinkedList constructor. Any modifications made through the List view change the array, and any modifications made to the array change the List view. The only operation permitted on the view returned by asList is *set*, which changes the value of the view and the backing array. Any other attempts to change the view (such as adding or removing elements) result in an **UnsupportedOperationException**.

Viewing Arrays as Lists *and Converting* Lists *to Arrays*

Figure J.4 uses Arrays method asList to view an array as a List and uses **List method toArray** to get an array from a LinkedList collection. The program calls method asList to create a List view of an array, which is used to initialize a LinkedList object, then adds

```
 1  // Fig. J.4: UsingToArray.java
 2  // Viewing arrays as Lists and converting Lists to arrays.
 3  import java.util.LinkedList;
 4  import java.util.Arrays;
 5
 6  public class UsingToArray
 7  {
 8     // creates a LinkedList, adds elements and converts to array
 9     public static void main( String[] args )
10     {
11        String[] colors = { "black", "blue", "yellow" };
12
13        LinkedList< String > links =
14           new LinkedList< String >( Arrays.asList( colors ) );
15
16        links.addLast( "red" ); // add as last item
17        links.add( "pink" ); // add to the end
18        links.add( 3, "green" ); // add at 3rd index
19        links.addFirst( "cyan" ); // add as first item
20
21        // get LinkedList elements as an array
22        colors = links.toArray( new String[ links.size() ] );
23
24        System.out.println( "colors: " );
25
26        for ( String color : colors )
27           System.out.println( color );
28     } // end main
29  } // end class UsingToArray
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

Fig. J.4 | Viewing arrays as `Lists` and converting `Lists` to arrays.

a series of strings to the `LinkedList` and calls method `toArray` to obtain an array containing references to the `Strings`.

Lines 13–14 construct a `LinkedList` of `Strings` containing the elements of array `colors`. Line 14 uses `Arrays` method `asList` to return a `List` view of the array, then uses that to initialize the `LinkedList` with its constructor that receives a `Collection` as an argument (a `List` *is a* `Collection`). Line 16 calls **`LinkedList` method `addLast`** to add `"red"` to the end of `links`. Lines 17–18 call **`LinkedList` method `add`** to add `"pink"` as the last element and `"green"` as the element at index `3` (i.e., the fourth element). Method `addLast` (line 16) functions identically to method `add` (line 17). Line 19 calls **`LinkedList` method `addFirst`** to add `"cyan"` as the new first item in the `LinkedList`. The `add` operations are permitted because they operate on the `LinkedList` object, not the view returned by `asList`.

Line 22 calls the `List` interface's `toArray` method to get a `String` array from `links`. The array is a copy of the list's elements—modifying the array's contents does *not* modify the list. The array passed to method `toArray` is of the same type that you'd like method `toArray` to return. If the number of elements in that array is greater than or equal to the number of elements in the `LinkedList`, `toArray` copies the list's elements into its array argument and returns that array. If the `LinkedList` has more elements than the number of elements in the array passed to `toArray`, `toArray` allocates a new array of the same type it receives as an argument, copies the list's elements into the new array and returns the new array.

J.6 Collections Methods

Class `Collections` provides several high-performance algorithms (Fig. J.5) for manipulating collection elements. The algorithms are implemented as `static` methods. The methods `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` and `copy` operate on `List`s. Methods `min`, `max` and `addAll` operate on `Collection`s.

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`.
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a `Collection`.

Fig. J.5 | Some methods of class `Collections`.

J.6.1 Method sort

Method `sort` sorts the elements of a `List`, which must implement the **`Comparable` interface**. The order is determined by the natural order of the elements' type as implemented by a `compareTo` method. Method `compareTo` is declared in interface `Comparable` and is sometimes called the **natural comparison method**. The `sort` call may specify as a second argument a **`Comparator`** object that determines an alternative ordering of the elements.

Sorting in Ascending or Descending Order

If `list` is a `List` of `Comparable` objects (such as `String`s), you can use `Collections` method `sort` to order the elements in ascending order as follows:

```
    Collections.sort( list ); // sort list into ascending order
```

You can sort the `List` in descending order as follows:

```
    // sort list into descending order
    Collections.sort( list, Collections.reverseOrder() );
```

The static **Collections method reverseOrder** returns a Comparator object that orders the collection's elements in reverse order.

Sorting with a **Comparator**

For objects that are not Comparable, you can create custom Comparators. Figure J.6 creates a custom Comparator class, named TimeComparator, that implements interface Comparator to compare two Time2 objects. Class Time2, declared in Fig. F.5, represents times with hours, minutes and seconds.

```java
 1  // Fig. J.8: TimeComparator.java
 2  // Custom Comparator class that compares two Time2 objects.
 3  import java.util.Comparator;
 4
 5  public class TimeComparator implements Comparator< Time2 >
 6  {
 7     public int compare( Time2 time1, Time2 time2 )
 8     {
 9        int hourCompare = time1.getHour() - time2.getHour(); // compare hour
10
11        // test the hour first
12        if ( hourCompare != 0 )
13           return hourCompare;
14
15        int minuteCompare =
16           time1.getMinute() - time2.getMinute(); // compare minute
17
18        // then test the minute
19        if ( minuteCompare != 0 )
20           return minuteCompare;
21
22        int secondCompare =
23           time1.getSecond() - time2.getSecond(); // compare second
24
25        return secondCompare; // return result of comparing seconds
26     } // end method compare
27  } // end class TimeComparator
```

Fig. J.6 | Custom Comparator class that compares two Time2 objects.

Class TimeComparator implements interface Comparator, a generic type that takes one type argument (in this case Time2). A class that implements Comparator must declare a compare method that receives two arguments and returns a negative integer if the first argument is less than the second, 0 if the arguments are equal or a positive integer if the first argument is greater than the second. Method compare (lines 7–26) performs comparisons between Time2 objects. Line 9 compares the two hours of the Time2 objects. If the hours are different (line 12), then we return this value. If this value is positive, then the first hour is greater than the second and the first time is greater than the second. If this value is negative, then the first hour is less than the second and the first time is less than the second. If this value is zero, the hours are the same and we must test the minutes (and maybe the seconds) to determine which time is greater.

Figure J.7 sorts a list using the custom Comparator class TimeComparator. Line 11 creates an ArrayList of Time2 objects. Recall that both ArrayList and List are generic types and accept a type argument that specifies the element type of the collection. Lines 13–17 create five Time2 objects and add them to this list. Line 23 calls method sort, passing it an object of our TimeComparator class (Fig. J.6).

```
1  // Fig. J.7: Sort.java
2  // Collections method sort with a custom Comparator object.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3
8  {
9     public static void main( String[] args )
10    {
11       List< Time2 > list = new ArrayList< Time2 >(); // create List
12
13       list.add( new Time2(  6, 24, 34 ) );
14       list.add( new Time2( 18, 14, 58 ) );
15       list.add( new Time2(  6, 05, 34 ) );
16       list.add( new Time2( 12, 14, 58 ) );
17       list.add( new Time2(  6, 24, 22 ) );
18
19       // output List elements
20       System.out.printf( "Unsorted array elements:\n%s\n", list );
21
22       // sort in order using a comparator
23       Collections.sort( list, new TimeComparator() );
24
25       // output List elements
26       System.out.printf( "Sorted list elements:\n%s\n", list );
27    } // end main
28 } // end class Sort3
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Fig. J.7 | Collections method sort with a custom Comparator object.

J.6.2 Method shuffle

Method **shuffle** randomly orders a List's elements. Appendix E presented a card shuffling and dealing simulation that shuffled a deck of cards with a loop. If you have an array of 52 Card objects, you can shuffle them with method shuffle as follows:

```
List< Card > list = Arrays.asList( deck ); // get List
Collections.shuffle( list ); // shuffle deck
```

The second line above shuffles the array by calling static method shuffle of class Collections. Method shuffle requires a List argument, so we must obtain a List view of the array before we can shuffle it. The `Arrays` class's `static` method `asList` gets a `List` view of the `deck` array.

J.7 Interface Queue

A queue is a collection that represents a waiting line—typically, insertions are made at the back of a queue and deletions are made from the front. Interface **`Queue`** extends interface `Collection` and provides additional operations for inserting, removing and inspecting elements in a queue. You can view the details of interface `Queue` and the list of classes that implement it at

```
docs.oracle.com/javase/6/docs/api/index.html?java/util/Queue.html
```

J.8 Sets

A **`Set`** is an unordered `Collection` of unique elements (i.e., no duplicate elements). The collections framework contains several `Set` implementations, including **`HashSet`** and **`TreeSet`**. `HashSet` stores its elements in a hash table, and `TreeSet` stores its elements in a tree. Hash tables are presented in Section J.9.

Figure J.8 uses a `HashSet` to remove duplicate strings from a `List`. Recall that both `List` and `Collection` are generic types, so line 16 creates a `List` that contains `String` objects, and line 20 passes a `Collection` of `String`s to method `printNonDuplicates`.

```
 1  // Fig. J.8: SetTest.java
 2  // HashSet used to remove duplicate values from an array of strings.
 3  import java.util.List;
 4  import java.util.Arrays;
 5  import java.util.HashSet;
 6  import java.util.Set;
 7  import java.util.Collection;
 8
 9  public class SetTest
10  {
11     public static void main( String[] args )
12     {
13        // create and display a List< String >
14        String[] colors = { "red", "white", "blue", "green", "gray",
15           "orange", "tan", "white", "cyan", "peach", "gray", "orange" };
16        List< String > list = Arrays.asList( colors );
17        System.out.printf( "List: %s\n", list );
18
19        // eliminate duplicates then print the unique values
20        printNonDuplicates( list );
21     } // end main
22
23     // create a Set from a Collection to eliminate duplicates
24     private static void printNonDuplicates( Collection< String > values )
25     {
26        // create a HashSet
27        Set< String > set = new HashSet< String >( values );
```

Fig. J.8 | `HashSet` used to remove duplicate values from an array of strings. (Part 1 of 2.)

```
28
29         System.out.print( "\nNonduplicates are: " );
30
31         for ( String value : set )
32            System.out.printf( "%s ", value );
33
34         System.out.println();
35      } // end method printNonDuplicates
36   } // end class SetTest
```

```
List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray,
orange]

Nonduplicates are: orange green white peach gray cyan red blue tan
```

Fig. J.8 | `HashSet` used to remove duplicate values from an array of strings. (Part 2 of 2.)

Method `printNonDuplicates` (lines 24–35) takes a `Collection` argument. Line 27 constructs a `HashSet<String>` from the `Collection<String>` argument. By definition, `Set`s do not contain duplicates, so when the `HashSet` is constructed, it removes any duplicates in the `Collection`. Lines 31–32 output elements in the `Set`.

Sorted Sets

The collections framework also includes the **`SortedSet` interface** (which extends `Set`) for sets that maintain their elements in sorted order—either the elements' natural order (e.g., numbers are in ascending order) or an order specified by a `Comparator`. Class `TreeSet` implements `SortedSet`. Items placed in a TreeSet are sorted as they're added.

J.9 Maps

`Map`s associate keys to values. The keys in a `Map` must be unique, but the associated values need not be. If a `Map` contains both unique keys and unique values, it's said to implement a **one-to-one mapping**. If only the keys are unique, the `Map` is said to implement a **many-to-one mapping**—many keys can map to one value.

`Map`s differ from `Set`s in that `Map`s contain keys and values, whereas `Set`s contain only values. Three of the several classes that implement interface `Map` are **`Hashtable`**, **`HashMap`** and **`TreeMap`**, and maps are used extensively in Android. `Hashtable`s and `HashMap`s store elements in hash tables, and `TreeMap`s store elements in trees—the details of the underlying data structures are beyond the scope of this book. **Interface `SortedMap`** extends `Map` and maintains its keys in sorted order—either the elements' natural order or an order specified by a `Comparator`. Class `TreeMap` implements `SortedMap`. Figure J.9 uses a `HashMap` to count the number of occurrences of each word in a string.

```
// Fig. J.9: WordTypeCount.java
// Program counts the number of occurrences of each word in a String.
import java.util.Map;
```

Fig. J.9 | Program counts the number of occurrences of each word in a `String`. (Part 1 of 3.)

```
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount
{
   public static void main( String[] args )
   {
      // create HashMap to store String keys and Integer values
      Map< String, Integer > myMap = new HashMap< String, Integer >();

      createMap( myMap ); // create map based on user input
      displayMap( myMap ); // display map content
   } // end main

   // create map from user input
   private static void createMap( Map< String, Integer > map )
   {
      Scanner scanner = new Scanner( System.in ); // create scanner
      System.out.println( "Enter a string:" ); // prompt for user input
      String input = scanner.nextLine();

      // tokenize the input
      String[] tokens = input.split( " " );

      // processing input text
      for ( String token : tokens )
      {
         String word = token.toLowerCase(); // get lowercase word

         // if the map contains the word
         if ( map.containsKey( word ) ) // is word in map
         {
            int count = map.get( word ); // get current count
            map.put( word, count + 1 ); // increment count
         } // end if
         else
            map.put( word, 1 ); // add new word with a count of 1 to map
      } // end for
   } // end method createMap

   // display map content
   private static void displayMap( Map< String, Integer > map )
   {
      Set< String > keys = map.keySet(); // get keys

      // sort keys
      TreeSet< String > sortedKeys = new TreeSet< String >( keys );

      System.out.println( "\nMap contains:\nKey\t\tValue" );

```

Fig. J.9 | Program counts the number of occurrences of each word in a `String`. (Part 2 of 3.)

```
56        // generate output for each key in map
57        for ( String key : sortedKeys )
58           System.out.printf( "%-10s%10s\n", key, map.get( key ) );
59
60        System.out.printf(
61           "\nsize: %d\nisEmpty: %b\n", map.size(), map.isEmpty() );
62     } // end method displayMap
63  } // end class WordTypeCount
```

```
Enter a string:
this is a sample sentence with several words this is another sample
sentence with several different words

Map contains:
Key            Value
a                  1
another            1
different          1
is                 2
sample             2
sentence           2
several            2
this               2
with               2
words              2

size: 10
isEmpty: false
```

Fig. J.9 | Program counts the number of occurrences of each word in a `String`. (Part 3 of 3.)

Line 14 creates an empty `HashMap` with a default initial capacity (16 elements) and a default load factor (0.75)—these defaults are built into the implementation of `HashMap`. When the number of occupied slots in the `HashMap` becomes greater than the capacity times the load factor, the capacity is doubled automatically. `HashMap` is a generic class that takes two type arguments—the type of key (i.e., `String`) and the type of value (i.e., `Integer`). Recall that the type arguments passed to a generic class must be reference types, hence the second type argument is `Integer`, not `int`.

Line 16 calls method `createMap` (lines 21–44), which uses a `map` to store the number of occurrences of each word in the sentence. Line 25 obtains the user input, and line 28 tokenizes it. The loop in lines 31–43 converts the next token to lowercase letters (line 33), then calls **`Map` method `containsKey`** (line 36) to determine whether the word is in the map (and thus has occurred previously in the string). If the `Map` does not contain a mapping for the word, line 42 uses **`Map` method `put`** to create a new entry in the map, with the word as the key and an `Integer` object containing `1` as the value. Autoboxing occurs when the program passes integer `1` to method `put`, because the map stores the number of occurrences of the word as an `Integer`. If the word does exist in the map, line 38 uses **`Map` method `get`** to obtain the key's associated value (the count) in the map. Line 39 increments that value and uses `put` to replace the key's associated value in the map. Method `put` returns the key's prior associated value, or `null` if the key was not in the map.

Method `displayMap` (lines 47–62) displays all the entries in the map. It uses **`HashMap` method `keySet`** (line 49) to get a set of the keys. The keys have type `String` in the `map`, so method `keySet` returns a generic type `Set` with type parameter specified to be `String`. Line 52 creates a `TreeSet` of the keys, in which the keys are sorted. The loop in lines 57–58 accesses each key and its value in the map. Line 58 displays each key and its value using format specifier `%-10s` to left justify each key and format specifier `%10s` to right justify each value. The keys are displayed in ascending order. Line 61 calls **`Map` method `size`** to get the number of key/value pairs in the `Map`. Line 61 also calls **`Map` method `isEmpty`**, which returns a `boolean` indicating whether the `Map` is empty.

J.10 Introduction to Files and Streams

Data stored in variables and arrays is temporary—it's lost when a local variable goes out of scope or when the program terminates. For long-term retention of data, even after the programs that create the data terminate, computers use **files**. You use files every day for tasks such as writing a document or creating a spreadsheet. Data maintained in files is **persistent data**—it exists beyond the duration of program execution.

Files as Streams of Bytes

Java views each file as a sequential **stream of bytes** (Fig. J.10). Every operating system provides a mechanism to determine the end of a file, such as an **end-of-file marker** or a count of the total bytes in the file that's recorded in a system-maintained administrative data structure. A Java program processing a stream of bytes simply receives an indication from the operating system when it reaches the end of the stream—the program does *not* need to know how the underlying platform represents files or streams. In some cases, the end-of-file indication occurs as an exception. In other cases, the indication is a return value from a method invoked on a stream-processing object.

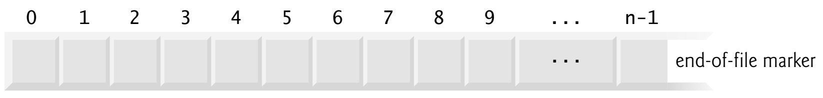


Fig. J.10 | Java's view of a file of *n* bytes.

Byte-Based and Character-Based Streams

Streams can be used to input and output data as bytes or characters. **Byte-based streams** input and output data in its binary format. **Character-based streams** input and output data as a sequence of characters. If the value `5` were being stored using a byte-based stream, it would be stored in the binary format of the numeric value `5`, or `101`. If the value `5` were being stored using a character-based stream, it would be stored in the binary format of the character `5`, or `00000000 00110101` (this is the binary representation for the numeric value `53`, which indicates the Unicode® character `5`). The difference between the two forms is that the numeric value can be used as an integer in calculations, whereas the character `5` is simply a character that can be used in a string of text, as in `"Sarah Miller is 15 years old"`. Files that are created using byte-based streams are referred to as **binary files**, while files created using character-based streams are referred to as **text files**. Text files can be read by text editors, while binary files are read by programs that understand the file's specific content and its ordering.

Opening a File

A Java program **opens** a file by creating an object and associating a stream of bytes or characters with it. The object's constructor interacts with the operating system to open the file.

The `java.io` Package

Java programs perform file processing by using classes from package **`java.io`**. This package includes definitions for stream classes, such as **`FileInputStream`** (for byte-based input from a file), **`FileOutputStream`** (for byte-based output to a file), **`FileReader`** (for character-based input from a file) and **`FileWriter`** (for character-based output to a file), which inherit from classes `InputStream`, `OutputStream`, `Reader` and `Writer`, respectively. Thus, the methods of the these stream classes can also be applied to file streams.

Java contains classes that enable you to perform input and output of objects or variables of primitive data types. The data will still be stored as bytes or characters behind the scenes, allowing you to read or write data in the form of `int`s, `String`s, or other types without having to worry about the details of converting such values to byte format. To perform such input and output, objects of classes **`ObjectInputStream`** and **`ObjectOutputStream`** can be used together with the byte-based file stream classes `FileInputStream` and `FileOutputStream` (these classes will be discussed in more detail shortly). The complete hierarchy of types in package `java.io` can be viewed in the online documentation at

```
    docs.oracle.com/javase/6/docs/api/java/io/package-tree.html
```

Character-based input and output can also be performed with classes `Scanner` and **`Formatter`**. Class `Scanner` is used extensively to input data from the keyboard—it can also read data from a file. Class `Formatter` enables formatted data to be output to any text-based stream in a manner similar to method `System.out.printf`.

J.11 Class `File`

Class **`File`** is useful for retrieving information about files or directories from disk. `File` objects are used frequently with objects of other `java.io` classes to specify files or directories to manipulate.

Creating `File` Objects

Class `File` provides several constructors. The one with a `String` argument specifies the `name` of a file or directory to associate with the `File` object. The `name` can contain **path information** as well as a file or directory name. A file or directory's path specifies its location on disk. The path includes some or all of the directories leading to the file or directory. An **absolute path** contains all the directories, starting with the **root directory**, that lead to a specific file or directory. Every file or directory on a particular disk drive has the same root directory in its path. A **relative path** normally starts from the directory in which the application began executing and is therefore "relative" to the current directory. The constructor with two `String` arguments specifies an absolute or relative path as the first argument and the file or directory to associate with the `File` object as the second argument. The constructor with `File` and `String` arguments uses an existing `File` object that specifies the parent directory of the file or directory specified by the `String` argument. The fourth constructor uses a `URI` object to locate the file. A **Uniform Resource Identifier** (**URI**) is a more general form of the **Uniform Resource Locators** (**URLs**) that are used to locate websites.

For example, `http://www.deitel.com/` is the URL for the Deitel & Associates website. URIs for locating files vary across operating systems. On Windows platforms, the URI

```
    file://C:/data.txt
```

identifies the file `data.txt` stored in the root directory of the C: drive. On UNIX/Linux platforms, the URI

```
    file:/home/student/data.txt
```

identifies the file `data.txt` stored in the `home` directory of the user `student`.

Figure J.11 lists some common `File` methods. The complete list can be viewed at `docs.oracle.com/javase/6/docs/api/java/io/File.html`.

Method	Description
`boolean canRead()`	Returns `true` if a file is readable by the current application; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writable by the current application; `false` otherwise.
`boolean exists()`	Returns `true` if the file or directory represented by the `File` object exists; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.
`String getAbsolutePath()`	Returns a `String` with the absolute path of the file or directory.
`String getName()`	Returns a `String` with the name of the file or directory.
`String getPath()`	Returns a `String` with the path of the file or directory.
`String getParent()`	Returns a `String` with the parent directory of the file or directory (i.e., the directory in which the file or directory is located).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, an unspecified value is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of `Strings` representing a directory's contents. Returns `null` if the `File` object does not represent a directory.

Fig. J.11 | `File` methods.

J.12 Introduction to Object Serialization

Java provides **object serialization** for writing entire objects to a stream and reading entire objects from a stream. A so-called **serialized object** is an object represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object. After a serialized object has been written into a file, it can be read from the file and **deserialized**—that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Classes `ObjectInputStream` and `ObjectOutputStream`

Classes `ObjectInputStream` and `ObjectOutputStream`, which respectively implement the **`ObjectInput`** and **`ObjectOutput`** interfaces, enable entire objects to be read from or written to a stream (possibly a file). To use serialization with files, we initialize `ObjectInputStream` and `ObjectOutputStream` objects with stream objects that read from and write to files—objects of classes `FileInputStream` and `FileOutputStream`, respectively. Initializing stream objects with other stream objects in this manner is sometimes called **wrapping**—the new stream object being created wraps the stream object specified as a constructor argument. To wrap a `FileInputStream` in an `ObjectInputStream`, for instance, we pass the `FileInputStream` object to the `ObjectInputStream`'s constructor.

Interfaces `ObjectOutput` and `ObjectInput`

The `ObjectOutput` interface contains method **`writeObject`**, which takes an `Object` as an argument and writes its information to an `OutputStream`. A class that implements interface `ObjectOutput` (such as `ObjectOutputStream`) declares this method and ensures that the object being output implements interface `Serializable` (discussed shortly). Correspondingly, the `ObjectInput` interface contains method **`readObject`**, which reads and returns a reference to an `Object` from an `InputStream`. After an object has been read, its reference can be cast to the object's actual type.

J.13 Introduction to Multithreading

It would be nice if we could focus our attention on performing only one action at a time and performing it well, but that's usually difficult to do. The human body performs a great variety of operations *in parallel*—or, as we say in programming, **concurrently**. Respiration, blood circulation, digestion, thinking and walking, for example, can occur concurrently, as can all the senses—sight, touch, smell, taste and hearing.

Computers, too, can perform operations concurrently. It's common for personal computers to compile a program, send a file to a printer and receive electronic mail messages over a network concurrently. Only computers that have multiple processors can truly execute multiple instructions concurrently. Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities, but on such computers only a single instruction can execute at once. Today's multicore computers have multiple processors that enable computers to perform tasks truly concurrently. Multicore smartphones are starting to appear.

Java Concurrency

Java makes concurrency available to you through the language and APIs. Java programs can have multiple **threads of execution**, where each thread has its own method-call stack and program counter, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory. This capability is called **multithreading**.

Performance Tip J.1

A problem with single-threaded applications that can lead to poor responsiveness is that lengthy activities must complete before *others can begin. In a multithreaded application, threads can be distributed across multiple processors (if available) so that multiple tasks execute* truly *concurrently and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems that* simulate *concurrency—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.*

Concurrent Programming Uses

We'll discuss many applications of **concurrent programming**. For example, when downloading a large file (e.g., an image, an audio clip or a video clip) over the Internet, the user may not want to wait until the entire clip downloads before starting the playback. To solve this problem, multiple threads can be used—one to download the clip, and another to play it. These activities proceed concurrently. To avoid choppy playback, the threads are **synchronized** (that is, their actions are coordinated) so that the player thread doesn't begin until there's a sufficient amount of the clip in memory to keep the player thread busy. The Java Virtual Machine (JVM) creates threads to run programs and threads to perform housekeeping tasks such as garbage collection.

Concurrent Programming Is Difficult

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought. To see why multithreaded programs can be difficult to write and understand, try the following experiment: Open three books to page 1, and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you'll appreciate many of the challenges of multithreading—switching between the books, reading briefly, remembering your place in each book, moving the book you're reading closer so that you can see it and pushing the books you're not reading aside—and, amid all this chaos, trying to comprehend the content of the books!

Use the Prebuilt Classes of the Concurrency APIs Whenever Possible

Programming concurrent applications is difficult and error prone. If you must use synchronization in a program, you should *use existing classes from the Concurrency APIs that manage synchronization for you*. These classes are written by experts, have been thoroughly tested and debugged, operate efficiently and help you avoid common traps and pitfalls.

J.14 Creating and Executing Threads with the Executor Framework

This section demonstrates how to perform concurrent tasks in an application by using `Executors` and `Runnable` objects.

Creating Concurrent Tasks with the* `Runnable` *Interface

You implement the **`Runnable`** interface (of package `java.lang`) to specify a task that can execute concurrently with other tasks. The `Runnable` interface declares the single method **`run`**, which contains the code that defines the task that a `Runnable` object should perform.

Executing **`Runnable`** *Objects with an* **`Executor`**

To allow a `Runnable` to perform its task, you must execute it. An **`Executor`** object executes `Runnable`s. An `Executor` does this by creating and managing a group of threads called a **thread pool**. When an `Executor` begins executing a `Runnable`, the `Executor` calls the `Runnable` object's `run` method, which executes in the new thread.

The `Executor` interface declares a single method named **`execute`** which accepts a `Runnable` as an argument. The `Executor` assigns every `Runnable` passed to its `execute` method to one of the available threads in the thread pool. If there are no available threads, the `Executor` creates a new thread or waits for a thread to become available and assigns that thread the `Runnable` that was passed to method `execute`.

Using an `Executor` has many advantages over creating threads yourself. `Executor`s can *reuse existing threads* to eliminate the overhead of creating a new thread for each task and can improve performance by *optimizing the number of threads* to ensure that the processor stays busy, without creating so many threads that the application runs out of resources.

Software Engineering Observation J.2

Though it's possible to create threads explicitly, it's recommended that you use the `Executor` interface to manage the execution of `Runnable` objects.

Using Class **`Executors`** *to Obtain an* **`ExecutorService`**

The **`ExecutorService` interface** (of package `java.util.concurrent`) *extends* `Executor` and declares various methods for managing the life cycle of an `Executor`. An object that implements the `ExecutorService` interface can be created using `static` methods declared in class **`Executors`** (of package `java.util.concurrent`). We use interface `ExecutorService` and a method of class `Executors` in our example, which executes three tasks.

Implementing the **`Runnable`** *Interface*

Class `PrintTask` (Fig. J.12) implements `Runnable` (line 5), *so that multiple `PrintTask`s can execute concurrently*. Variable `sleepTime` (line 7) stores a random integer value from 0 to 5 seconds created in the `PrintTask` constructor (line 17). Each thread running a `PrintTask` sleeps for the amount of time specified by `sleepTime`, then outputs its task's name and a message indicating that it's done sleeping.

```
 1  // Fig. J.12: PrintTask.java
 2  // PrintTask class sleeps for a random time from 0 to 5 seconds
 3  import java.util.Random;
 4
 5  public class PrintTask implements Runnable
 6  {
 7     private final int sleepTime; // random sleep time for thread
 8     private final String taskName; // name of task
 9     private final static Random generator = new Random();
10
11     // constructor
12     public PrintTask( String name )
13     {
14        taskName = name; // set task name
```

Fig. J.12 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 1 of 2.)

```
15
16          // pick random sleep time between 0 and 5 seconds
17          sleepTime = generator.nextInt( 5000 ); // milliseconds
18       } // end PrintTask constructor
19
20       // method run contains the code that a thread will execute
21       public void run()
22       {
23          try // put thread to sleep for sleepTime amount of time
24          {
25             System.out.printf( "%s going to sleep for %d milliseconds.\n",
26                taskName, sleepTime );
27             Thread.sleep( sleepTime ); // put thread to sleep
28          } // end try
29          catch ( InterruptedException exception )
30          {
31             System.out.printf( "%s %s\n", taskName,
32                "terminated prematurely due to interruption" );
33          } // end catch
34
35          // print task name
36          System.out.printf( "%s done sleeping\n", taskName );
37       } // end method run
38    } // end class PrintTask
```

Fig. J.12 | PrintTask class sleeps for a random time from 0 to 5 seconds. (Part 2 of 2.)

A PrintTask executes when a thread calls the PrintTask's run method. Lines 25–26 display a message indicating the name of the currently executing task and that the task is going to sleep for sleepTime milliseconds. Line 27 invokes static method **sleep** of class Thread to place the thread in the *timed waiting* state for the specified amount of time. At this point, the thread loses the processor, and the system allows another thread to execute. When the thread awakens, it reenters the *runnable* state. When the PrintTask is assigned to a processor again, line 36 outputs a message indicating that the task is done sleeping, then method run terminates. The catch at lines 29–33 is required because method sleep might throw a *checked* exception of type **InterruptedException** if a sleeping thread's **interrupt** method is called.

Using the **ExecutorService** *to Manage Threads that Execute* **PrintTasks**

Figure J.13 uses an ExecutorService object to manage threads that execute PrintTasks (as defined in Fig. J.12). Lines 11–13 create and name three PrintTasks to execute. Line 18 uses Executors method **newCachedThreadPool** to obtain an ExecutorService that's capable of creating new threads as they're needed by the application. These threads are used by ExecutorService (threadExecutor) to execute the Runnables.

```
1    // Fig. J.13: TaskExecutor.java
2    // Using an ExecutorService to execute Runnables.
3    import java.util.concurrent.Executors;
4    import java.util.concurrent.ExecutorService;
```

Fig. J.13 | Using an ExecutorService to execute Runnables. (Part 1 of 2.)

```
 5
 6  public class TaskExecutor
 7  {
 8     public static void main( String[] args )
 9     {
10        // create and name each runnable
11        PrintTask task1 = new PrintTask( "task1" );
12        PrintTask task2 = new PrintTask( "task2" );
13        PrintTask task3 = new PrintTask( "task3" );
14
15        System.out.println( "Starting Executor" );
16
17        // create ExecutorService to manage threads
18        ExecutorService threadExecutor = Executors.newCachedThreadPool();
19
20        // start threads and place in runnable state
21        threadExecutor.execute( task1 ); // start task1
22        threadExecutor.execute( task2 ); // start task2
23        threadExecutor.execute( task3 ); // start task3
24
25        // shut down worker threads when their tasks complete
26        threadExecutor.shutdown();
27
28        System.out.println( "Tasks started, main ends.\n" );
29     } // end main
30  } // end class TaskExecutor
```

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

```
Starting Executor
task1 going to sleep for 3161 milliseconds.
task3 going to sleep for 532 milliseconds.
task2 going to sleep for 3440 milliseconds.
Tasks started, main ends.

task3 done sleeping
task1 done sleeping
task2 done sleeping
```

Fig. J.13 | Using an `ExecutorService` to execute `Runnables`. (Part 2 of 2.)

Lines 21–23 each invoke the `ExecutorService`'s `execute` method, which executes the `Runnable` passed to it as an argument (in this case a `PrintTask`) some time in the future. The specified task may execute in one of the threads in the `ExecutorService`'s thread pool, in a new thread created to execute it, or in the thread that called the `execute` method—the `ExecutorService` manages these details. Method `execute` returns immediately from each invocation—the program does *not* wait for each PrintTask to finish. Line 26 calls ExecutorService method **shutdown**, which notifies the ExecutorService to *stop accepting new tasks, but continues executing tasks that have already been submitted.* Once all of the previously submitted Runnables have completed, the threadExecutor terminates. Line 28 outputs a message indicating that the tasks were started and the main thread is finishing its execution.

The code in main executes in the **main thread**, a thread created by the JVM. The code in the run method of PrintTask (lines 21–37 of Fig. J.12) executes whenever the Executor starts each PrintTask—again, this is sometime after they're passed to the ExecutorService's execute method (Fig. J.13, lines 21–23). When main terminates, the program itself continues running because there are still tasks that must finish executing. The program will not terminate until these tasks complete.

The sample outputs show each task's name and sleep time as the thread goes to sleep. The one with the shortest sleep time *normally* awakens first, indicates that it's done sleeping and terminates. In the first output, the main thread terminates *before* any of the PrintTasks output their names and sleep times. This shows that the main thread runs to completion before the PrintTasks get a chance to run. In the second output, all of the PrintTasks output their names and sleep times *before* the main thread terminates. Also, notice in the second example output, task3 goes to sleep before task2, even though we passed task2 to the ExecutorService's execute method before task3. This illustrates the fact that *we cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started.*

J.15 Overview of Thread Synchronization

When multiple threads share an object and it's modified by one or more of them, indeterminate results may occur unless access to the shared object is managed properly. If one thread is in the process of updating a shared object and another thread also tries to update it, it's unclear which thread's update takes effect. When this happens, the program's behavior cannot be trusted—sometimes the program will produce the correct results, and sometimes it won't. In either case, there'll be no indication that the shared object was manipulated incorrectly.

The problem can be solved by giving only one thread at a time *exclusive access* to code that manipulates the shared object. During that time, other threads desiring to manipulate the object are kept waiting. When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting is allowed to proceed. This process, called **thread synchronization**, coordinates access to shared data by multiple concurrent threads. By synchronizing threads in this manner, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called **mutual exclusion**.

Monitors

A common way to perform synchronization is to use Java's built-in **monitors**. Every object has a monitor and a **monitor lock** (or **intrinsic lock**). The monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time, and thus can be used to enforce mutual exclusion. If an operation requires the executing thread to hold a lock while the operation is performed, a thread must acquire the lock before proceeding with the operation. Other threads attempting to perform an operation that requires the same lock will be *blocked* until the first thread releases the lock, at which point the *blocked* threads may attempt to acquire the lock and proceed with the operation.

To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a **`synchronized` statement**. Such code is said to be **guarded** by the monitor lock; a thread must **acquire the lock** to execute the guarded statements. The monitor allows only one thread at a time to execute statements within `synchronized` statements that lock on the same object, as only one thread at a time can hold the monitor lock. The `synchronized` statements are declared using the **`synchronized` keyword**:

```
synchronized ( object )
{
   statements
} // end synchronized statement
```

where *object* is the object whose monitor lock will be acquired; *object* is normally `this` if it's the object in which the `synchronized` statement appears. If several `synchronized` statements are trying to execute on an object at the same time, only one of them may be active on the object—all the other threads attempting to enter a `synchronized` statement on the same object are temporarily *blocked* from executing.

When a `synchronized` statement finishes executing, the object's monitor lock is released and one of the *blocked* threads attempting to enter a `synchronized` statement can be allowed to acquire the lock to proceed. Java also allows **`synchronized` methods**. Before executing, a non-`static` `synchronized` method must acquire the lock on the object that's used to call the method. Similary, a `static` `synchronized` method must acquire the lock on the class that's used to call the method.

J.16 Concurrent Collections Overview

Earlier in this appendix, we introduced various collections from the Java Collections API. The collections from the `java.util.concurrent` package are specifically designed and optimized for use in programs that share collections among multiple threads. For information on the many concurrent collections in package `java.util.concurrent`, visit

```
docs.oracle.com/javase/6/docs/api/java/util/concurrent/
   package-summary.html
```

J.17 Multithreading with GUI

Swing applications present a unique set of challenges for multithreaded programming. All Swing applications have an **event dispatch thread** to handle interactions with the GUI components. Typical interactions include *updating GUI components* or *processing user actions* such as mouse clicks. All tasks that require interaction with an application's GUI are placed in an *event queue* and are executed sequentially by the event dispatch thread.

Swing GUI components are not thread safe—they cannot be manipulated by multiple threads without the risk of incorrect results. Thread safety in GUI applications is achieved not by synchronizing thread actions, but by *ensuring that Swing components are accessed from the event dispatch thread*—a technique called **thread confinement**.

Usually it's sufficient to perform simple tasks on the event dispatch thread in sequence with GUI component manipulations. If a lengthy task is performed in the event dispatch thread, it cannot attend to other tasks in the event queue while it's tied up in that task. This causes the GUI to become unresponsive. *Long-running tasks should be handled in separate threads*, freeing the event dispatch thread to continue managing other GUI interactions. Of course, to update the GUI based on the tasks's results, you must use the event dispatch thread, rather than from the worker thread that performed the computation.

Class `SwingWorker`

Class **`SwingWorker`** (in package `javax.swing`) perform long-running tasks in a worker thread and to update Swing components from the event dispatch thread based on the tasks' results. `SwingWorker` implements the `Runnable` interface, meaning that *a `SwingWorker` object can be scheduled to execute in a separate thread*. The `SwingWorker` class provides several methods to simplify performing tasks in a worker thread and making the results available for display in a GUI. Some common `SwingWorker` methods are described in Fig. J.14. Class `SwingWorker` is similar to class `AsyncTask`, which is used frequently in Android apps.

Method	Description
`doInBackground`	Defines a long task and is called in a worker thread.
`done`	Executes on the event dispatch thread when `doInBackground` returns.
`execute`	Schedules the `SwingWorker` object to be executed in a worker thread.
`get`	Waits for the task to complete, then returns the result of the task (i.e., the return value of `doInBackground`).
`publish`	Sends intermediate results from the `doInBackground` method to the `process` method for processing on the event dispatch thread.
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Fig. J.14 | Commonly used `SwingWorker` methods.

Performing Tasks in a Worker Thread

In the next example, the user enters a number *n* and the program gets the *n*th Fibonacci number, which we calculate using a recursive algorithm. The algorithm is time consuming for large values, so we use a `SwingWorker` object to perform the calculation in a worker thread. The GUI also allows the user to get the next Fibonacci number in the sequence with each click of a button, beginning with `fibonacci(1)`. This short calculation is performed directly in the event dispatch thread. The program is capable of producing up to the 92nd Fibonacci number—subsequent values are outside the range that can be represented by a `long`. You can use class `BigInteger` to represent arbitrarily large integer values.

Class `BackgroundCalculator` (Fig. J.15) performs the recursive Fibonacci calculation in a *worker thread*. This class extends `SwingWorker` (line 8), overriding the methods `doInBackground` and `done`. Method `doInBackground` (lines 21–24) computes the *n*th Fibonacci number in a worker thread and returns the result. Method `done` (lines 27–43) displays the result in a `JLabel`.

```java
// Fig. J.15: BackgroundCalculator.java
// SwingWorker subclass for calculating Fibonacci numbers
// in a worker thread.
import javax.swing.SwingWorker;
import javax.swing.JLabel;
import java.util.concurrent.ExecutionException;

public class BackgroundCalculator extends SwingWorker< Long, Object >
{
   private final int n; // Fibonacci number to calculate
   private final JLabel resultJLabel; // JLabel to display the result

   // constructor
   public BackgroundCalculator( int number, JLabel label )
   {
      n = number;
      resultJLabel = label;
   } // end BackgroundCalculator constructor

   // long-running code to be run in a worker thread
   public Long doInBackground()
   {
      return nthFib = fibonacci( n );
   } // end method doInBackground

   // code to run on the event dispatch thread when doInBackground returns
   protected void done()
   {
      try
      {
         // get the result of doInBackground and display it
         resultJLabel.setText( get().toString() );
      } // end try
      catch ( InterruptedException ex )
      {
         resultJLabel.setText( "Interrupted while waiting for results." );
      } // end catch
      catch ( ExecutionException ex )
      {
         resultJLabel.setText(
            "Error encountered while performing calculation." );
      } // end catch
   } // end method done

   // recursive method fibonacci; calculates nth Fibonacci number
   public long fibonacci( long number )
   {
      if ( number == 0 || number == 1 )
         return number;
      else
         return fibonacci( number - 1 ) + fibonacci( number - 2 );
   } // end method fibonacci
} // end class BackgroundCalculator
```

Fig. J.15 | `SwingWorker` subclass for calculating Fibonacci numbers in a worker thread.

`SwingWorker` is a *generic class*. In line 8, the first type parameter is `Long` and the second is `Object`. The first type parameter indicates the type returned by the `doInBackground` method; the second indicates the type that's passed between the `publish` and `process` methods to handle intermediate results. Since we do not use `publish` and `process` in this example, we simply use `Object` as the second type parameter.

A `BackgroundCalculator` object can be instantiated from a class that controls a GUI. A `BackgroundCalculator` maintains instance variables for an integer that represents the Fibonacci number to be calculated and a `JLabel` that displays the results of the calculation (lines 10–11). The `BackgroundCalculator` constructor (lines 14–18) initializes these instance variables with the arguments that are passed to the constructor.

Software Engineering Observation J.3

Any GUI components that will be manipulated by `SwingWorker` methods, such as components that will be updated from methods `process` or `done`, should be passed to the `SwingWorker` subclass's constructor and stored in the subclass object. This gives these methods access to the GUI components they'll manipulate.

When method `execute` is called on a `BackgroundCalculator` object, the object is scheduled for execution in a worker thread. Method `doInBackground` is called from the worker thread and invokes the `fibonacci` method (lines 46–52), passing instance variable `n` as an argument (line 23). Method `fibonacci` uses recursion to compute the Fibonacci of `n`. When `fibonacci` returns, method `doInBackground` returns the result.

After `doInBackground` returns, method `done` is automatically called from the event dispatch thread. This method attempts to set the result `JLabel` to the return value of `doInBackground` by calling method `get` to retrieve this return value (line 32). Method `get` waits for the result to be ready if necessary, but since we call it from method `done`, the computation will be complete before `get` is called. Lines 34–37 catch `InterruptedException` if the current thread is interrupted while waiting for `get` to return. This exception will not occur in this example since the calculation will have already completed by the time `get` is called. Lines 38–42 catch `ExecutionException`, which is thrown if an exception occurs during the computation.

Class `FibonacciNumbers`

Class `FibonacciNumbers` (Fig. J.16) displays a window containing two sets of GUI components—one set to compute a Fibonacci number in a worker thread and another to get the next Fibonacci number in response to the user's clicking a `JButton`. The constructor (lines 38–109) places these components in separate titled `JPanels`. Lines 46–47 and 78–79 add two `JLabels`, a `JTextField` and a `JButton` to the `workerJPanel` to allow the user to enter an integer whose Fibonacci number will be calculated by the `BackgroundWorker`. Lines 84–85 and 103 add two `JLabels` and a `JButton` to the event dispatch thread panel to allow the user to get the next Fibonacci number in the sequence. Instance variables `n1` and `n2` contain the previous two Fibonacci numbers in the sequence and are initialized to `0` and `1`, respectively (lines 29–30). Instance variable `count` stores the most recently computed sequence number and is initialized to `1` (line 31). The two `JLabels` display `count` and `n2` initially, so that the user will see the text `Fibonacci of 1: 1` in the `eventThreadJPanel` when the GUI starts.

```java
// Fig. J.16: FibonacciNumbers.java
// Using SwingWorker to perform a long calculation with
// results displayed in a GUI.
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.border.TitledBorder;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.util.concurrent.ExecutionException;

public class FibonacciNumbers extends JFrame
{
   // components for calculating the Fibonacci of a user-entered number
   private final JPanel workerJPanel =
      new JPanel( new GridLayout( 2, 2, 5, 5 ) );
   private final JTextField numberJTextField = new JTextField();
   private final JButton goJButton = new JButton( "Go" );
   private final JLabel fibonacciJLabel = new JLabel();

   // components and variables for getting the next Fibonacci number
   private final JPanel eventThreadJPanel =
      new JPanel( new GridLayout( 2, 2, 5, 5 ) );
   private long n1 = 0; // initialize with first Fibonacci number
   private long n2 = 1; // initialize with second Fibonacci number
   private int count = 1; // current Fibonacci number to display
   private final JLabel nJLabel = new JLabel( "Fibonacci of 1: " );
   private final JLabel nFibonacciJLabel =
      new JLabel( String.valueOf( n2 ) );
   private final JButton nextNumberJButton = new JButton( "Next Number" );

   // constructor
   public FibonacciNumbers()
   {
      super( "Fibonacci Numbers" );
      setLayout( new GridLayout( 2, 1, 10, 10 ) );

      // add GUI components to the SwingWorker panel
      workerJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "With SwingWorker" ) );
      workerJPanel.add( new JLabel( "Get Fibonacci of:" ) );
      workerJPanel.add( numberJTextField );
      goJButton.addActionListener(
         new ActionListener()
         {
```

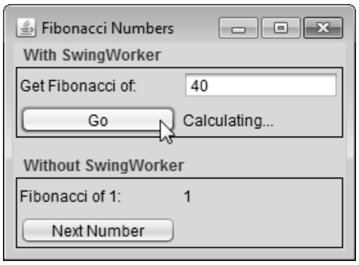
Fig. J.16 | Using SwingWorker to perform a long calculation with results displayed in a GUI. (Part 1 of 3.)

```
            public void actionPerformed( ActionEvent event )
            {
               int n;

               try
               {
                  // retrieve user's input as an integer
                  n = Integer.parseInt( numberJTextField.getText() );
               } // end try
               catch( NumberFormatException ex )
               {
                  // display an error message if the user did not
                  // enter an integer
                  fibonacciJLabel.setText( "Enter an integer." );
                  return;
               } // end catch

               // indicate that the calculation has begun
               fibonacciJLabel.setText( "Calculating..." );

               // create a task to perform calculation in background
               BackgroundCalculator task =
                  new BackgroundCalculator( n, fibonacciJLabel );
               task.execute(); // execute the task
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
      workerJPanel.add( goJButton );
      workerJPanel.add( fibonacciJLabel );

      // add GUI components to the event-dispatching thread panel
      eventThreadJPanel.setBorder( new TitledBorder(
         new LineBorder( Color.BLACK ), "Without SwingWorker" ) );
      eventThreadJPanel.add( nJLabel );
      eventThreadJPanel.add( nFibonacciJLabel );
      nextNumberJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed( ActionEvent event )
            {
               // calculate the Fibonacci number after n2
               long temp = n1 + n2;
               n1 = n2;
               n2 = temp;
               ++count;

               // display the next Fibonacci number
               nJLabel.setText( "Fibonacci of " + count + ": " );
               nFibonacciJLabel.setText( String.valueOf( n2 ) );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
```

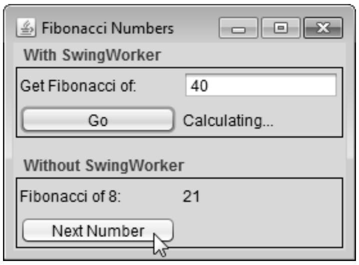
Fig. J.16 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 2 of 3.)

```
      eventThreadJPanel.add( nextNumberJButton );

      add( workerJPanel );
      add( eventThreadJPanel );
      setSize( 275, 200 );
      setVisible( true );
   } // end constructor

   // main method begins program execution
   public static void main( String[] args )
   {
      FibonacciNumbers application = new FibonacciNumbers();
      application.setDefaultCloseOperation( EXIT_ON_CLOSE );
   } // end main
} // end class FibonacciNumbers
```

a) Begin calculating Fibonacci of 40 in the background



b) Calculating other Fibonacci values while Fibonacci of 40 continues calculating



c) Fibonacci of 40 calculation finishes

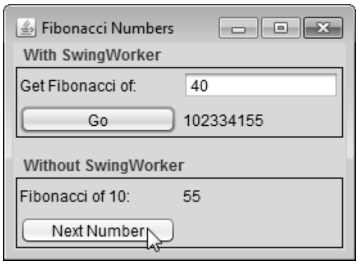


Fig. J.16 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 3 of 3.)

Lines 48–77 register the event handler for the `goJButton`. If the user clicks this `JButton`, line 58 gets the value entered in the `numberJTextField` and attempts to parse it as an integer. Lines 72–73 create a new `BackgroundCalculator` object, passing in the user-entered value and the `fibonacciJLabel` that's used to display the calculation's results. Line 74 calls method `execute` on the `BackgroundCalculator`, scheduling it for execution in a separate worker thread. Method `execute` does not wait for the `BackgroundCalculator` to finish executing. It returns immediately, allowing the GUI to continue processing other events while the computation is performed.

If the user clicks the nextNumberJButton in the eventThreadJPanel, the event handler registered in lines 86–102 executes. Lines 92–95 add the previous two Fibonacci numbers stored in n1 and n2 to determine the next number in the sequence, update n1 and n2 to their new values and increment count. Then lines 98–99 update the GUI to display the next number. The code for these calculations is in method actionPerformed, so they're performed on the *event dispatch thread*. Handling such short computations in the event dispatch thread does not cause the GUI to become unresponsive, as with the recursive algorithm for calculating the Fibonacci of a large number. Because the longer Fibonacci computation is performed in a separate worker thread using the SwingWorker, it's possible to get the next Fibonacci number while the recursive computation is still in progress.

J.18 Wrap-Up

In this appendix, you used classes ArrayList and LinkedList, which both implement the List interface. You used several predefined methods for manipulating collections. Next, you learned how to use the Set interface and class HashSet to manipulate an unordered collection of unique values. We discussed the SortedSet interface and class TreeSet for manipulating a sorted collection of unique values. You then learned about Java's interfaces and classes for manipulating key/value pairs—Map, SortedMap, HashMap and TreeMap. We discussed the Collections class's static methods for obtaining unmodifiable and synchronized views of collections.

Next, we introduced fundamental concepts of file and stream processing and overviewed object serialization. Finally, we introduced multithreading. You learned that Java makes concurrency available to you through the language and APIs. You also learned that the JVM itself creates threads to run a program, and that it also can create threads to perform housekeeping tasks such as garbage collection. We presented the interface Runnable, which is used to specify a task that can execute concurrently with other tasks. We showed how to use the Executor interface to manage the execution of Runnable objects via thread pools, which can reuse existing threads to eliminate the overhead of creating a new thread for each task and can improve performance by optimizing the number of threads to ensure that the processor stays busy. We discussed how to use a synchronized block to coordinate access to shared data by multiple concurrent threads.

We discussed the fact that Swing GUIs are not thread safe, so all interactions with and modifications to the GUI must be performed in the event dispatch thread. We also discussed the problems associated with performing long-running calculations in the event dispatch thread. Then we showed how you can use the SwingWorker class to perform long-running calculations in worker threads and how to display the results of a SwingWorker in a GUI when the calculation completed.

Self-Review Exercises

J.1 Fill in the blanks in each of the following statements:

a) A(n) ________ is used to iterate through a collection and can remove elements from the collection during the iteration.

b) An element in a List can be accessed by using the element's ________.

c) Assuming that myArray contains references to Double objects, ________ occurs when the statement "myArray[0] = 1.25;" executes.

d) Java classes ________ and ________ provide the capabilities of arraylike data structures that can resize themselves dynamically.
e) Assuming that `myArray` contains references to `Double` objects, ________ occurs when the statement "`double number = myArray[ 0 ];`" executes.
f) `ExecutorService` method ________ ends each thread in an `ExecutorService` as soon as it finishes executing its current `Runnable`, if any.
g) Keyword ________ indicates that only one thread at a time should execute on an object.

J.2 Determine whether each statement is *true* or *false*. If *false*, explain why.
a) Values of primitive types may be stored directly in a collection.
b) A `Set` can contain duplicate values.
c) A `Map` can contain duplicate keys.
d) A `List` is an ordered collection that can contain duplicate elements.
e) `Collections` is an interface.
f) `Iterators` can remove elements.
g) Method `exists` of class `File` returns `true` if the name specified as the argument to the `File` constructor is a file or directory in the specified path.
h) Binary files are human readable in a text editor.
i) An absolute path contains all the directories, starting with the root directory, that lead to a specific file or directory.

Answers to Self-Review Exercises

J.1 a) `Iterator`. b) index. c) autoboxing. d) `ArrayList`, `Vector`. e) auto-unboxing. f) `shutdown`. g) `synchronized`.

J.2 a) False. Autoboxing occurs when adding a primitive type to a collection, which means the primitive type is converted to its corresponding type-wrapper class. b) False. A `Set` cannot contain duplicate values. c) False. A `Map` cannot contain duplicate keys. d) True. e) False. `Collections` is a class; `Collection` is an interface. f) True. g) True. h) False. Text files are human readable in a text editor. Binary files might be human readable, but only if the bytes in the file represent ASCII characters. i) True.

Execises

J.3 Define each of the following terms:
a) `Collection`
b) `Collections`
c) `Comparator`
d) `List`
e) `HashMap`
f) `ObjectOutputStream`
g) `File`
h) `ObjectOutputStream`
i) byte-based stream
j) character-based stream

J.4 Briefly answer the following questions:
a) What is the primary difference between a `Set` and a `List`?
b) Explain any two types of `Lists`.
c) Can you print all the elements in a collection without using an `Iterator`? If yes, how?

J.5 ***(Duplicate Elimination)*** Write a program that reads in a series of first names and eliminates duplicates by storing them in a `Set`. Allow the user to search for a first name.

J.6 *(Counting Letters)* Modify the program of Fig. J.9 to count the number of occurrences of each letter rather than of each word. For example, the string `"HELLO THERE"` contains two `H`s, three `E`s, two `L`s, one `O`, one `T` and one `R`. Display the results.

J.7 *(Color Chooser)* Use a `HashMap` to create a reusable class for choosing one of the 13 predefined colors in class `Color`. The names of the colors should be used as keys, and the predefined `Color` objects should be used as values. Place this class in a package that can be imported into any Java program. Use your new class in an application that allows the user to select a color and draw a shape in that color.

J.8 *(Counting Duplicate Words)* Write a program that determines and prints the number of duplicate words in a sentence. Treat uppercase and lowercase letters the same. Ignore punctuation.

J.9 *(Prime Numbers and Prime Factors)* Write a program that takes a whole number input from a user and determines whether it's prime. If the number is not prime, display its unique prime factors. Remember that a prime number's factors are only 1 and the prime number itself. Every number that is not prime has a unique prime factorization. For example, consider the number 54. The prime factors of 54 are 2, 3, 3 and 3. When the values are multiplied together, the result is 54. For the number 54, the prime factors output should be 2 and 3. Use `Set`s as part of your solution.

J.10 *(Sorting Words with a `TreeSet`)* Write a program that uses a `String` method `split` to tokenize a line of text input by the user and places each token in a `TreeSet`. Print the elements of the `TreeSet`. [*Note:* This should cause the elements to be printed in ascending sorted order.]

J.11 *(Bouncing Ball)* Write a program that bounces a blue ball inside a `JPanel`. The ball should begin moving with a `mousePressed` event. When the ball hits the edge of the `JPanel`, it should bounce off the edge and continue in the opposite direction. The ball should be updated using a `Runnable`.

K

Operator Precedence Chart

Operators are shown in decreasing order of precedence from top to bottom (Fig. K.1).

Operator	Description	Associativity
`++` `--`	unary postfix increment unary postfix decrement	right to left

`++`
`--`
`+`
`-`
`!`
`~`
`(` *type* `)`
	unary prefix increment
	unary prefix decrement
	unary plus
	unary minus
	unary logical negation
	unary bitwise complement
	unary cast
		right to left

`*`
`/`
`%`
	multiplication
	division
	remainder
		left to right

`+` `-`	addition or string concatenation subtraction	left to right

`<<`
`>>`
`>>>`
	left shift
	signed right shift
	unsigned right shift
		left to right

`<`
`<=`
`>`
`>=`
`instanceof`
	less than
	less than or equal to
	greater than
	greater than or equal to
	type comparison
		left to right

`==`
`!=`
	is equal to
	is not equal to
		left to right

`&`	bitwise AND boolean logical AND	left to right
`^`	bitwise exclusive OR boolean logical exclusive OR	left to right

Fig. K.1 | Operator precedence chart. (Part 1 of 2.)

Operator	Description	Associativity
`|`	bitwise inclusive OR	left to right
	boolean logical inclusive OR	
&&	conditional AND	left to right
`||`	conditional OR	left to right
?:	conditional	right to left
=	assignment	right to left
+=	addition assignment	
-=	subtraction assignment	
*=	multiplication assignment	
/=	division assignment	
%=	remainder assignment	
&=	bitwise AND assignment	
^=	bitwise exclusive OR assignment	
`|=`	bitwise inclusive OR assignment	
<<=	bitwise left-shift assignment	
>>=	bitwise signed-right-shift assignment	
>>>=	bitwise unsigned-right-shift assignment	

Fig. K.1 | Operator precedence chart. (Part 2 of 2.)

L

Primitive Types

Type	Size in bits	Values	Standard
`boolean`		`true` or `false`	
[*Note:* A `boolean`'s representation is specific to the Java Virtual Machine on each platform.]			
`char`	16	`'\u0000'` to `'\uFFFF'` (0 to 65535)	(ISO Unicode character set)
`byte`	8	–128 to +127 ($–2^7$ to $2^7 – 1$)	
`short`	16	–32,768 to +32,767 ($–2^{15}$ to $2^{15} – 1$)	
`int`	32	–2,147,483,648 to +2,147,483,647 ($–2^{31}$ to $2^{31} – 1$)	
`long`	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 ($–2^{63}$ to $2^{63} – 1$)	
`float`	32	*Negative range:* –3.4028234663852886E+38 to –1.40129846432481707e–45 *Positive range:* 1.40129846432481707e–45 to 3.4028234663852886E+38	(IEEE 754 floating point)
`double`	64	*Negative range:* –1.7976931348623157E+308 to –4.94065645841246544e–324 *Positive range:* 4.94065645841246544e–324 to 1.7976931348623157E+308	(IEEE 754 floating point)

Fig. L.1 | Java primitive types.

For more information on IEEE 754 visit `grouper.ieee.org/groups/754/`.

Index